Byte Ordering for Integer Binary Data on Big Endian and Little Endian Platforms  
Working with Packed Decimal and Zoned Decimal Data  
Informs by Category  
Dictionary  

Chapter 6 △ SAS ARM Macros  
Definition of ARM Macros  
Using ARM Macros  
Defining User Metrics in ARM Macros  
Defining Correlators in ARM Macros  
Enabling ARM Macro Execution  
Setting the Macro Environment  
Using ARM Post-Processing Macros  
Troubleshooting Error Messages  
ARM Macros by Category  
Dictionary  

Chapter 7 △ Statements  
Definition of Statements  
DATA Step Statements  
Global Statements  
Dictionary  

Chapter 8 △ SAS System Options  
Definition of System Options  
Syntax  
Using SAS System Options  
Comparisons  
SAS System Options by Category  
Dictionary  

PART 2   
Appendixes  

Appendix 1 △ DATA Step Object Attributes and Methods  
The DATA Step Component Object Interface  
Dot Notation and DATA Step Component Objects  
Dictionary  

Appendix 2 △ DATA Step Debugger  
Introduction  
Basic Usage  
Advanced Usage: Using the Macro Facility with the Debugger  
Examples  
Commands  
Dictionary
What’s New

Overview

New and enhanced features in Base SAS save you time, effort, and system resources by providing faster processing and easier data access and management, more robust analysis, and improved data presentation.

- By using new SAS system options that enable threading and the use of multiple CPUs, the following SAS procedures take advantage of multi-processing I/O: SORT, SQL, MEANS, TABULATE, and REPORT.
- Two pre-defined component objects for the DATA step enable you to quickly store, search, and retrieve data based on lookup keys.
- The FILENAME statement now supports directory services, multiple FTP service commands, and Secure Sockets Layering (SSL).
- The Application Response Measurement (ARM) system enables you to monitor the availability and performance of transactions within and across diverse applications.
- The Perl regular expression (PRX) functions and CALL routines use a modified version of Perl as a pattern-matching language to enhance search-and-replace operations on text.
- New character functions search and compare character strings in addition to concatenating character strings.
- There are several new descriptive statistic functions and mathematical functions.
- New formats, informats, and functions support international and local values for money, datetime, and Unicode values. All data set options, formats, informats, functions, and system options that relate to national language support are documented in the new SAS National Language Support (NLS): User’s Guide.
- A new ODS statement enables you to render multiple ODS output formats without re-running a PROC or a DATA step. See the SAS Output Delivery System: User’s Guide.
What's New

Note:
- This section describes the features of Base SAS that are new or enhanced since SAS 8.2.
- z/OS is the successor to the OS/390 operating system. SAS 9.1 (and later) is supported on both OS/390 and z/OS operating systems and, throughout this document, any reference to z/OS also applies to OS/390, unless otherwise stated.

---

SAS System Features

Application Response Measurement (ARM)

Application Response Measurement (ARM) enables you to monitor the availability and performance of transactions within and across diverse applications. The SAS ARM interface consists of the implementation of the ARM API as ARM macros and an ARM agent. An ARM agent generates calls to the ARM macros. New ARM system options enable you to manage the ARM environment and to log internal SAS processing transactions. See “Monitoring Performance Using Application Response Measurement (ARM)” in *SAS Language Reference: Concepts*, “ARM Macros” on page xviii, and “System Options” on page xx.

Cross-Environment Data Access (CEDA)

CEDA processes SAS files that were created on a different host. This is especially useful if you have upgraded from a 32–bit platform to a 64–bit platform. Messages in the SAS log notify you when CEDA is being used to process a SAS file. See “Processing Data Using Cross-Environment Data Access (CEDA)” in *SAS Language Reference: Concepts*.

DATA Step Object Attributes and Methods

SAS now provides two pre-defined component objects for use in a DATA step: the hash object and the hash iterator object. These objects enable you to quickly and efficiently store, search, and retrieve data based on lookup keys.

The DATA step component object interface enables you to create and manipulate these component objects by using statements, attributes, and methods. You use the DATA step object dot notation to access the component object’s attributes and methods. The hash and hash iterator objects have one attribute, fourteen methods, and two statements associated with them. See Appendix 1, “DATA Step Object Attributes and Methods,” on page 1635.

Engines

- The default BASE engine in SAS supports longer format and informat names, thread-enabled procedures such as the SORT and SUMMARY procedures, and more than 32,767 variables in a SAS data set.
- The metadata LIBNAME engine enables you to use metadata in order to access and augment data that is identified by the metadata. The metadata engine
retrieves information about the target SAS data library from metadata objects in a specified SAS Metadata Repository on the SAS Metadata Server. The metadata engine provides a consistent method for accessing many data sources. That is, SAS provides different engines that have different options, behavior, and tuning requirements. By taking advantage of metadata, the necessary information that is required to access data can be created in one central location so that applications can use the metadata engine to access different sources of data, without having to understand the differences and details of each SAS engine. See the SAS Metadata LIBNAME Engine: User's Guide.

- The XML LIBNAME engine imports and exports a broader variety of XML documents. The XMLMAP= option specifies a separate XML document that contains specific XMLMap syntax. The XMLMap syntax, Version 1.2, tells the XML engine how to interpret the XML markup in order to successfully import an XML document. See the SAS XML LIBNAME Engine User's Guide.


- The new character variable padding (CVP) engine expands character variable lengths, using a specified expansion amount, so that character data truncation does not occur when a file requires transcoding. Character data truncation can occur when the number of bytes for a character in one encoding is different from the number of bytes for the same character in another encoding, such as when a single-byte character set (SBCS) is transcoded to a double-byte character set (DBCS). See the SAS National Language Support (NLS): User's Guide.

### Indexing

When creating an index that requires sorting, SAS tries to sort the data by using the thread-enabled sort. By dividing the sorting task into separately executable processes, the time that is required to sort the data can be reduced. See the topic “Creating an Index” in Understanding SAS Indexes in SAS Language Reference: Concepts.

### Integrity Constraints

Variables in a SAS data file can now be part of both a primary key (general integrity constraint) and a foreign key (referential integrity constraint). However, there are restrictions when defining a primary key constraint and a foreign key constraint that use the same variables. See the topic “Overlapping Primary Key and Foreign Key Constraints” in Understanding Integrity Constraints in SAS Language Reference: Concepts.

### Restricted System Options

System administrators can restrict system options from being modified by a user. You can use the RESTRICT option in the OPTIONS procedure to list the restricted options. The implementation of restricted options is specific to the operating environment. For details about how to restrict options, see the configuration guide for your operating environment. For information about listing restricted options, see the OPTIONS procedure in the Base SAS Procedures Guide.
SAS Utility Macro

The SAS utility macro, %DS2CSV, is available now in Base SAS. This macro converts SAS data sets to comma-separated values (CSV) files. Prior to SAS 9.1, this macro was available only for SAS/IntrNet users.

Universal Unique Identifiers

A Universal Unique Identifier (UUID) is a 128-bit identifier that consists of date and time information, and the IEEE node address of a host. UUIDs are useful when objects such as rows or other components of a SAS application must be uniquely identified. For more information, see “Universal Unique Identifiers” in SAS Language Reference: Concepts.

SAS Language Elements


Data Set Options

- The following data set options are new:
  
  **OBSBUF=**
  determines the size of the view buffer for processing a DATA step view.
  
  **SPILL=**
  specifies whether to create a spill file for non-sequential processing of a DATA step view.

- The following data set options are enhanced:
  
  **BUFNO=**
  supports the same syntax as the BUFNO= system option in order to specify the number of buffers to be allocated for processing a SAS data set.
  
  **BUFSIZE=**
  supports the same syntax as the BUFSIZE= system option in order to specify the permanent buffer page size for an output SAS data set.
  
  **FIRSTOBS=**
  supports the same syntax as the FIRSTOBS= system option in order to specify which observation SAS processes first.
  
  **OBS=**
  supports the same syntax as the OBS= system option in order to specify when to stop processing observations.

Formats

- The maximum length for character format names is increased to 31. The maximum length for numeric format names is increased to 32.
The following formats are new:

**MMYY**
writes date values in the form \textit{mmMyyyy}, where M is the separator and the
year is written in either 2 or 4 digits.

**YYMM**
writes date values in the form \textit{yyMmm}, where the year is written in either
2 or 4 digits and M is the separator.

**YYQ**
writes date values in the form \textit{yyyyQq}, where the year is written in either 2
or 4 digits, Q is the separator, and q is the quarter of the year.

**YYQR**
writes date values in the form \textit{yyyyQqr}, where the year is written in either 2
or 4 digits, Q is the separator, and qr is the quarter of the year expressed in
Roman numerals.

The PVALUE format now returns missing values that are specified by the
MISSING= system option.

---

**Functions and CALL Routines**

New functions and CALL routines include character, mathematical, descriptive
statistical, and special functions, and character-string matching functions that can use
PERL expressions.

The following character functions are new:

**ANYALNUM**
searches a character string for an alphanumeric character and returns the
first position at which it is found.

**ANYALPHA**
searches a character string for an alphabetic character and returns the first
position at which it is found.

**ANYCNTRL**
searches a character string for a control character and returns the first
position at which it is found.

**ANYDIGIT**
searches a character string for a digit and returns the first position at which
it is found.

**ANYFIRST**
searches a character string for a character that is valid as the first character
in a SAS variable name under VALIDVARNAME=V7, and returns the first
position at which that character is found.

**ANYGRAPH**
searches a character string for a graphical character and returns the first
position at which it is found.

**ANYLOWER**
searches a character string for a lowercase letter and returns the first
position at which it is found.

**ANYNAME**
searches a character string for a character that is valid in a SAS variable
name under VALIDVARNAME=V7, and returns the first position at which
that character is found.
ANYPRINT
   searches a character string for a printable character and returns the first
   position at which it is found.

ANYPUNCT
   searches a character string for a punctuation character and returns the first
   position at which it is found.

ANYSPACE
   searches a character string for a white-space character (blank, horizontal tab,
   vertical tab, carriage return, line feed, or form feed), and returns the first
   position at which it is found.

ANYUPPER
   searches a character string for an uppercase letter and returns the first
   position at which it is found.

ANYXDIGIT
   searches a character string for a hexadecimal character that represents a
digit and returns the first position at which that character is found.

CAT
   concatenates character strings without removing leading or trailing blanks.

CATS
   concatenates character strings and removes leading and trailing blanks.

CATT
   concatenates character strings and removes trailing blanks only.

CATX
   concatenates character strings, removes leading and trailing blanks, and
   inserts separators.

CHOOSEC
   returns a character value that represents the results of choosing from a list of
   arguments.

CHOSEN
   returns a numeric value that represents the results of choosing from a list of
   arguments.

COMPARE
   returns the position of the left-most character by which two strings differ, or
   returns 0 if there is no difference.

COMPGED
   compares two strings by computing the generalized edit distance.

COMPLEV
   compares two strings by computing the Levenshtein edit distance.

COUNT
   counts the number of times that a specific substring of characters appears
   within a character string that you specify.

COUNTC
   counts the number of specific characters that either appear or do not appear
   within a character string that you specify.

FIND
   searches for a specific substring of characters within a character string that
   you specify.
FINDC
searches for specific characters that either appear or do not appear within a character string that you specify.

IFC
returns a character value that matches an expression.

IFN
returns a numeric value that matches an expression.

LENGTHC
returns the length of a character string, including trailing blanks.

LENGTHM
returns the amount of memory (in bytes) that is allocated for a character string.

LENGTHN
returns the length of a non-blank character string, excluding trailing blanks, and returns 0 for a blank character string.

NLITERAL
converts a character string that you specify to a SAS name literal (n-literal).

NOTALNUM
searches a character string for a non-alphanumeric character and returns the first position at which it is found.

NOTALPHA
searches a character string for a non-alphabetic character and returns the first position at which it is found.

NOTCNTRL
searches a character string for a character that is not a control character and returns the first position at which it is found.

NOTDIGIT
searches a character string for any character that is not a digit and returns the first position at which that character is found.

NOTFIRST
searches a character string for an invalid first character in a SAS variable name under VALIDVARNAME=V7, and returns the first position at which that character is found.

NOTGRAPH
searches a character string for a non-graphical character and returns the first position at which it is found.

NOTLOWER
searches a character string for a character that is not a lowercase letter and returns the first position at which that character is found.

NOTNAME
searches a character string for an invalid character in a SAS variable name under VALIDVARNAME=V7, and returns the first position at which that character is found.

NOTPRINT
searches a character string for a non-printable character and returns the first position at which it is found.
NOTPUNCT
searches a character string for a character that is not a punctuation character and returns the first position at which it is found.

NOTSPACE
searches a character string for a character that is not a white-space character (blank, horizontal tab, vertical tab, carriage return, line feed, or form feed), and returns the first position at which it is found.

NOTUPPER
searches a character string for a character that is not an uppercase letter and returns the first position at which that character is found.

NOTXDIGIT
searches a character string for a character that is not a hexadecimal digit and returns the first position at which that character is found.

NVALID
checks a character string for validity for use as a SAS variable name in a SAS statement.

PROPCASE
converts all words in an argument to proper case.

PRXCHANGE
performs a pattern-matching replacement.

PRXPOSN
returns the value for a capture buffer.

SCANQ
returns the \( n \)th word from a character expression and ignores delimiters that are enclosed in quotation marks.

STRIP
returns a character string with all leading and trailing blanks removed.

SUBPAD
returns a substring that has a length you specify, using blank padding if necessary.

SUBSTRN
returns a substring that allows a result with a length of 0.

The following descriptive statistics functions are new:

GEOMEAN
returns the geometric mean.

GEOMEANZ
returns the geometric mean without fuzzing the values of the arguments that are approximately 0.

HARMEAN
returns the harmonic mean.

HARMEANZ
returns the harmonic mean without fuzzing the values of the arguments that are approximately 0.

IQR
returns the interquartile range.
LARGEST
returns the \( k \)th largest non-missing value.

MAD
returns the median absolute deviation from the median.

MEDIAN
computes median values.

MODZ
returns the remainder from the division of the first argument by the second argument; uses 0 fuzzing.

PCTL
computes percentiles.

RMS
returns the root mean square.

SMALLEST
returns the \( k \)th smallest non-missing value.

The following External Files function is new:

DCREATE
creates an external directory.

The following macro functions are new:

SYMEXIST
indicates the existence of a macro variable.

SYMGLOBL
indicates whether a macro variable has global scope in the DATA step during DATA step execution.

SYMLOCAL
indicates whether a macro variable has local scope in the DATA step during DATA step execution.

The following mathematical functions are new:

BETA
returns the value of the beta function.

COALESCE
returns the first non-missing value from a list of numeric arguments.

COALESCEC
returns the first non-missing value from a list of character arguments.

LOGBETA
returns the logarithm of the beta function.

The following probability function is new:

LOGCDF
returns the logarithm of a left cumulative distribution function.

The following quantile function is new:

QUANTILE
returns the quantile from the specified distribution.

The following special function is new:

UUIDGEN
returns the short or the binary form of a Universal Unique Identifier (UUID).
The following state and ZIP code function is new:

\texttt{ZIPCITY}

returns a city name and the two-character postal code that corresponds to a ZIP code.

The following trigonometric function is new:

\texttt{ATAN2}

returns the arc tangent of two numeric variables.

The following truncation functions are new:

\texttt{CEILZ}

returns the smallest integer that is greater than or equal to the argument; uses 0 fuzzing.

\texttt{FLOORZ}

returns the largest integer that is less than or equal to the argument; uses 0 fuzzing.

\texttt{INTZ}

returns the integer portion of the argument; uses 0 fuzzing.

\texttt{ROUND}

rounds the first argument to the nearest multiple of the second argument, or to the nearest integer when the second argument is omitted.

\texttt{ROUNDE}

rounds the first argument to the nearest multiple of the second argument, and returns an even multiple when the first argument is halfway between the two nearest multiples.

\texttt{ROUNDZ}

rounds the first argument to the nearest multiple of the second argument; uses 0 fuzzing.

The following variable information functions are new:

\texttt{VVALUE}

returns the formatted value that is associated with the variable that you specify.

\texttt{VVALUEX}

returns the formatted value that is associated with the argument that you specify.

Using Perl regular expression (PRX) functions and CALL routines is new. The following PRX functions are new. For more information, see “Pattern Matching Using SAS Regular Expressions (RX) and Perl Regular Expressions (PRX)” on page 260.

\texttt{PRXMATCH}

searches for a pattern match and returns the position at which the pattern is found.

\texttt{PRXPAREN}

returns the last bracket match for which there is a match in a pattern.

\texttt{PRXPARSE}

compiles a Perl regular expression that can be used for pattern-matching a character value.

\texttt{CALL PRXCHANGE}

performs a pattern-matching substitution.
CALL PRXDEBUG
enables Perl regular expressions in a DATA step to send debug output to the SAS log.

CALL PRXFREE
frees unneeded memory that was allocated for a Perl regular expression.

CALL PRXNEXT
returns the position and length of a substring that matches a pattern and iterates over multiple matches within one string.

CALL PRXPOSN
returns the start position and length for a capture buffer.

CALL PRXSUBSTR
returns the position and length of a substring that matches a pattern.

The following CALL routines are new:

CALL ALLPERM
generates all permutations of the values of several variables.

CALL CATS
concatenates character strings and removes leading and trailing blanks.

CALL CATT
concatenates character strings and removes trailing blanks only.

CALL CATX
concatenates character strings, removes leading and trailing blanks, and inserts separators.

CALL COMPCOST
sets the costs of operations for later use by the COMPGED function.

CALL LOGISTIC
returns the logistic value of each argument.

CALL MISSING
assigns a missing value to specified character or numeric variables.

CALL RANPERK
randomly permutes the values of the arguments and returns a permutation of $k$ out of $n$ values.

CALL RANPERM
randomly permutes the values of the arguments.

CALL SCAN
returns the position and length of a given word in a character expression.

CALL SCANQ
returns the position and length of a given word in a character expression, and ignores delimiters that are enclosed in quotation marks.

CALL SOFTMAX
returns the softmax value for each argument.

CALL STDIZE
standardizes the values of one or more variables.

CALL STREAMINIT
specifies a seed value to use for subsequent random number generation by the RAND function.
CALL SYMPUTX
assigns a value to a macro variable and removes both leading and trailing blanks.

CALL TANH
returns the hyperbolic tangent of each argument.

CALL VNEXT
returns the name, type, and length of a variable that is used in a DATA step.

The following functions are enhanced:

COMPRESS
accepts a third optional argument that can modify the characters in the second argument.

EXIST
accepts all SAS data library type members. A third optional argument enables you to specify a generation data set number.

INDEXW
accepts a third optional argument that enables you to use delimiters for inter-word boundaries.

accepts an argument that enables you to use a Perl regular expression.

SUBSTR (left of=)
assigns a length of 8 to an undeclared variable when the function is compiled.

Informats

The maximum length for character informat names is increased to 30. The maximum length for numeric informat names is increased to 31.

The following informats are new:

ANYDTDTE
extracts date values from DATE, DATETIME, DDMMYY, JULIAN, MMDDYY, MONYY, TIME, or YYQ informat values.

ANYDTEXTM
extracts datetime values from DATE, DATETIME, DDMMYY, JULIAN, MMDDYY, MONYY, TIME, or YYQ informat values.

ANYDTTME
extracts time values from DATE, DATETIME, DDMMYY, JULIAN, MMDDYY, MONYY, TIME, or YYQ informat values.

STIMERw.
reads time values and determines whether the values are hours, minutes, or seconds; reads the output of the STIMER system option.

ARM Macros

%ARMCNV, the new ARM macro, converts an ARM log that is created in SAS 9.0 and later, which uses a simple format, into the ARM log format that is used in SAS 8.2, which is more detailed.

SAS Utility Macro

%DS2CSV, the new SAS utility macro, converts SAS data sets to comma-separated values (CSV) files.
The following statements are new:

**ODS Statements**
control different features of the Output Delivery System. For more information about these statements, see the *SAS Output Delivery System: User’s Guide*.

**FILENAME, CLIPBAORD Access Method**
enables you to read text data from and write text data to the clipboard on the host machine.

**DECLARE**
declares a DATA step component object; creates an instance of and initializes data for a DATA step component object.

_**NEW**_
creates an instance of a DATA step component object.

**PUTLOG**
writes a message to the SAS log.

The following statements are enhanced:

**FILENAME, FTP Access Method**
supports directory services and multiple FTP service commands.

**FILENAME, URL Access Method**
supports Secure Sockets Layering (SSL).

**LIBNAME statement**
the following options are new:

**COMPRESS=**
controls the compression of observations in output SAS data sets in a SAS data library.

**CVPBYTES=**
specifies the number of bytes to use in order to expand character variable lengths when processing a SAS data set that requires transcoding.

**CVPENGINE=**
specifies which engine to use in order to process character variable lengths in a SAS data set that requires transcoding.

**CVPMULTIPLIER=**
specifies the multiplier value to use in order to expand character variable lengths when processing a SAS data set that requires transcoding.

**INENCODING=**
overrides the encoding for input processing.

**OUTENCODING=**
overrides the encoding for output processing.
System Options

The following system options are new:

ARMAGENT=
specifies an ARM agent, which is an executable module that contains a vendor’s implementation of the ARM API.

ARMLOC=
specifies the location of the ARM log.

ARMSUBSYS=
enables and disables the ARM subsystems that determine which internal SAS processing transactions should be logged.

AUTHPROVIDERDOMAIN=
associates a domain suffix with an authentication provider.

AUTOSAVELOC=
specifies the location of the Program Editor autosave file.

BYSORTED
specifies whether observations in one or more data sets are sorted in alphabetical or numerical order or are grouped in another logical order.

CMPLIB=
specifies one or more SAS catalogs that contain compiler subroutines that should be included during program compilation.

CMPOPT=
specifies which type of code generation optimizations should be used in the SAS language compiler.

CPUCOUNT=
specifies the number of processors that the thread-enabled applications should assume are available for concurrent processing.

DMSLOGSIZE=
specifies the maximum number of rows that can be displayed in the Log window in the SAS windowing environment.

DMSOUTSIZE=
specifies the maximum number of rows that can be displayed in the Output window in the SAS windowing environment.

DMSSYNCHK
enables syntax checking for multiple steps in the SAS windowing environment.

DTRESET
updates the date and the time in the SAS log and in the listing file.

EMAILAUTHPROTOCOL=
specifies the authentication protocol for SMTP e-mail.

EMAILID=
specifies the identity of the individual who is sending e-mail from within SAS.

EMAILPW=
specifies your e-mail login password.

ERRORBYABEND
specifies how SAS responds to BY-group error conditions.
FONTSLOC=
    specifies the location that contains the SAS fonts that are loaded by a printer
to use with Universal Printing.

HELPENCMD
    specifies whether SAS uses the English version or the translated version of
the keyword list for the command-line Help.

IBUFSIZE=
    specifies the buffer page size for an index file.

LOGPARM=
    controls when SAS log files are opened and closed and (in conjunction with
the LOG= system option) how they are named.

METAAUTORESOURCES=
    identifies which resources should be assigned at SAS initialization.

METACONNECT=
    identifies which named connection from the metadata user profiles should be
used as the default value for logging into the SAS Metadata Server.

METAENCRYPTALT=
    specifies which type of encryption should be used when communicating with a
SAS Metadata Server.

METAENCRYPTLEVEL=
    specifies what should be encrypted when communicating with a SAS
Metadata Server.

METAID=
    identifies the current SAS version that is installed on the SAS Metadata
Server.

METAPASSWORD=
    specifies the default password for the SAS Metadata Server.

METAPORT=
    specifies the TCP port for the SAS Metadata Server.

METAPROFILE=
    specifies which file contains the SAS Metadata Server user profiles.

METAPROTOCOL=
    specifies which network protocol should be used for communicating with the
SAS Metadata Server.

METAREPOSITORY=
    specifies which default SAS Metadata Repository should be used on the SAS
Metadata Server.

METASERVER=
    specifies the address of the SAS Metadata Server.

METAUSER=
    specifies the default user ID for logging on to the SAS Metadata Server.

PAGEBREAKINITIAL
    begins the SAS log and listing files on a new page.

QUOTELENMAX
    specifies that SAS write a warning to the SAS log about the maximum length
that can be used for strings that are enclosed in quotation marks.
SORTEQUALS  
controls the order in which PROC SORT arranges observations that have identical BY values in the output data set.

SYSPRINTFONT  
specifies the font for the current default printer.

SYNTAXCHECK  
specifies whether to validate SAS program syntax.

TERMSTMT=  
specifies which SAS statements should be executed when the SAS session is terminated.

TEXTURELOC=  
specifies the location of textures and images that are used by ODS styles.

THREADS  
specifies that SAS use threaded processing if it is available.

TOOLSMENU  
specifies whether to include or to suppress the Tools menu in windows that display SAS menus.

UUIDCOUNT  
specifies the number of UUIDs that should be acquired each time the UUID Generator Daemon is used.

UUIDGENHOST  
identifies the host and the port for the UUID Generator Daemon.

UTILLOC=  
specifies a set of file system locations in which applications can store utility files.

VALIDFMTNAME=  
specifies the length of format and informat names that can be used when creating new SAS data sets and format catalogs.

VIEWMENU  
specifies whether to include or to suppress the View menu in windows that display menus.

V6CREATEUPDATE=  
controls or monitors the creation of new, version 6 SAS data sets or the updating of existing, version 6 SAS data sets.

The following system options have been enhanced:

CMPOPT=  
specifies which type of code generation optimizations should be used in the SAS language compiler.

SORTSIZE=  
specifies the amount of memory that is available when using the SORT procedure.
Dictionary of Language Elements

Chapter 1. Introduction to the SAS 9.1 Language Reference: Dictionary 3

Chapter 2. SAS Data Set Options 5

Chapter 3. Formats 67

Chapter 4. Functions and CALL Routines 243

Chapter 5. Informats 929

Chapter 6. SAS ARM Macros 1053

Chapter 7. Statements 1087

Chapter 8. SAS System Options 1439
The SAS Language Reference: Dictionary

SAS Language Reference: Dictionary provides detailed reference information for the major language elements of Base SAS software:
- data set options
- formats
- functions and CALL routines
- informats
- Application Response Measurement (ARM) macros
- statements
- SAS system options.

It also includes the following four appendixes:
- DATA Step Object Attributes and Methods
- ENCODING= values for SAS commands and statements
- DATA step debugger
- Recommended reading.

For extensive conceptual information about the SAS System and the SAS language, including the DATA step, see SAS Language Reference: Concepts.
# CHAPTER 2

## SAS Data Set Options

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Definition of Data Set Options</td>
<td>6</td>
</tr>
<tr>
<td>Syntax</td>
<td>6</td>
</tr>
<tr>
<td>Using Data Set Options</td>
<td>6</td>
</tr>
<tr>
<td>Using Data Set Options with Input or Output SAS Data Sets</td>
<td>6</td>
</tr>
<tr>
<td>How Data Set Options Interact with System Options</td>
<td>7</td>
</tr>
<tr>
<td>Data Set Options by Category</td>
<td>7</td>
</tr>
<tr>
<td>Dictionary</td>
<td>9</td>
</tr>
<tr>
<td>ALTER= Data Set Option</td>
<td>9</td>
</tr>
<tr>
<td>BUFNO= Data Set Option</td>
<td>10</td>
</tr>
<tr>
<td>BUFSIZE= Data Set Option</td>
<td>12</td>
</tr>
<tr>
<td>CNTLLEV= Data Set Option</td>
<td>13</td>
</tr>
<tr>
<td>COMPRESS= Data Set Option</td>
<td>14</td>
</tr>
<tr>
<td>DLDMGACTION= Data Set Option</td>
<td>16</td>
</tr>
<tr>
<td>DROP= Data Set Option</td>
<td>17</td>
</tr>
<tr>
<td>ENCODING= Data Set Option</td>
<td>18</td>
</tr>
<tr>
<td>ENCRYPT= Data Set Option</td>
<td>18</td>
</tr>
<tr>
<td>FILECLOSE= Data Set Option</td>
<td>20</td>
</tr>
<tr>
<td>FIRSTOBS= Data Set Option</td>
<td>20</td>
</tr>
<tr>
<td>GENMAX= Data Set Option</td>
<td>22</td>
</tr>
<tr>
<td>GENNUM= Data Set Option</td>
<td>23</td>
</tr>
<tr>
<td>IDXNAME= Data Set Option</td>
<td>25</td>
</tr>
<tr>
<td>IDXWHERE= Data Set Option</td>
<td>26</td>
</tr>
<tr>
<td>IN= Data Set Option</td>
<td>27</td>
</tr>
<tr>
<td>INDEX= Data Set Option</td>
<td>28</td>
</tr>
<tr>
<td>KEEP= Data Set Option</td>
<td>30</td>
</tr>
<tr>
<td>LABEL= Data Set Option</td>
<td>31</td>
</tr>
<tr>
<td>OBS= Data Set Option</td>
<td>32</td>
</tr>
<tr>
<td>OBSBUF= Data Set Option</td>
<td>38</td>
</tr>
<tr>
<td>OUTREP= Data Set Option</td>
<td>40</td>
</tr>
<tr>
<td>POINTOBS= Data Set Option</td>
<td>42</td>
</tr>
<tr>
<td>PW= Data Set Option</td>
<td>43</td>
</tr>
<tr>
<td>PWREQ= Data Set Option</td>
<td>44</td>
</tr>
<tr>
<td>READ= Data Set Option</td>
<td>44</td>
</tr>
<tr>
<td>RENAME= Data Set Option</td>
<td>45</td>
</tr>
<tr>
<td>REPEMPTY= Data Set Option</td>
<td>47</td>
</tr>
<tr>
<td>REPLACE= Data Set Option</td>
<td>48</td>
</tr>
<tr>
<td>REUSE= Data Set Option</td>
<td>49</td>
</tr>
<tr>
<td>SORTEDBY= Data Set Option</td>
<td>50</td>
</tr>
<tr>
<td>SORTSEQ= Data Set Option</td>
<td>52</td>
</tr>
<tr>
<td>SPILL= Data Set Option</td>
<td>52</td>
</tr>
<tr>
<td>TOBSNO= Data Set Option</td>
<td>60</td>
</tr>
</tbody>
</table>
Definition of Data Set Options

_Data set options_ specify actions that apply only to the SAS data set with which they appear. They let you perform such operations as

- renaming variables
- selecting only the first or last \( n \) observations for processing
- dropping variables from processing or from the output data set
- specifying a password for a data set.

Syntax

Specify a data set option in parentheses after a SAS data set name. To specify several data set options, separate them with spaces.

\[(option-1=value-1<...option-n=value-n>)\]

These examples show data set options in SAS statements:

- `data scores(keep=team game1 game2 game3);`
- `proc print data=new(drop=year);`
- `set old(rename=(date=Start_Date));`

Using Data Set Options

Using Data Set Options with Input or Output SAS Data Sets

Most SAS data set options can apply to either input or output SAS data sets in DATA steps or procedure (PROC) steps. If a data set option is associated with an input data set, the action applies to the data set that is being read. If the option appears in the DATA statement or after an output data set specification in a PROC step, SAS applies the action to the output data set. In the DATA step, data set options for output data sets must appear in the DATA statement, not in any OUTPUT statements that may be present.

Some data set options, such as COMPRESS=, are meaningful only when you create a SAS data set because they set attributes that exist for the life of the data set. To change or cancel most data set options, you must re-create the data set. You can change other options (such as PW= and LABEL=) with PROC DATASETS. For more information, see the “DATASETS Procedure” in _Base SAS Procedures Guide_.

When data set options appear on both input and output data sets in the same DATA or PROC step, SAS applies data set options to input data sets before it evaluates programming statements or before it applies data set options to output data sets. Likewise, data set options that are specified for the data set being created are applied after programming statements are processed. For example, when using the RENAME= data set option, the new names are not associated with the variables until the DATA step ends.
In some instances, data set options conflict when they are used in the same statement. For example, you cannot specify both the DROP= and KEEP= data set options for the same variable in the same statement. Timing can also be an issue in some cases. For example, if using KEEP= and RENAME= on a data set specified in the SET statement, KEEP= needs to use the original variable names, because SAS will process KEEP= before the data set is read. The new names specified in RENAME= will apply to the programming statements that follow the SET statement.

### How Data Set Options Interact with System Options

Many system options and data set options share the same name and have the same function. System options remain in effect for all DATA and PROC steps in a SAS job or session unless they are respecified. The data set option overrides the system option for the data set in the step in which it appears. In this example, the OBS= system option in the OPTIONS statement specifies that only the first 100 observations will be processed from any data set within the SAS job. The OBS= data set option in the SET statement, however, overrides the system option for data set TWO and specifies that only the first 5 observations will be read from data set TWO. The PROC PRINT step prints the data set FINAL. This data set contains the first 5 observations from data set TWO, followed by the first 100 observations from data set THREE:

```sas
options obs=100;

data final;
  set two(obs=5) three;
run;

proc print data=final;
run;
```

### Data Set Options by Category

<table>
<thead>
<tr>
<th>Category</th>
<th>SAS Data Set Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Set Control</td>
<td>“ALTER= Data Set Option” on page 9</td>
<td>Assigns an alter password to a SAS file and enables access to a password-protected SAS file</td>
</tr>
<tr>
<td></td>
<td>“BUFNO= Data Set Option” on page 10</td>
<td>Specifies the number of buffers to be allocated for processing a SAS data set</td>
</tr>
<tr>
<td></td>
<td>“BUFSIZE= Data Set Option” on page 12</td>
<td>Specifies the permanent buffer page size for an output SAS data set</td>
</tr>
<tr>
<td></td>
<td>“CNTLLEV= Data Set Option” on page 13</td>
<td>Specifies the level of shared access to SAS data sets</td>
</tr>
<tr>
<td></td>
<td>“COMPRESS= Data Set Option” on page 14</td>
<td>Controls the compression of observations in an output SAS data set</td>
</tr>
<tr>
<td></td>
<td>“DLDMGACTION= Data Set Option” on page 16</td>
<td>Specifies what type of action to take when a SAS data set in a SAS data library is detected as damaged</td>
</tr>
<tr>
<td>Category</td>
<td>SAS Data Set Option</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Data Set Options by Category</td>
<td>“ENCODING= Data Set Option” on page 18</td>
<td>Overrides the encoding for processing a specific input or output SAS data set</td>
</tr>
<tr>
<td></td>
<td>“ENCRYPT= Data Set Option” on page 18</td>
<td>Encrypts SAS data files</td>
</tr>
<tr>
<td></td>
<td>“GENMAX= Data Set Option” on page 22</td>
<td>Requests generations for a data set and specifies the maximum number of versions</td>
</tr>
<tr>
<td></td>
<td>“GENNUM= Data Set Option” on page 23</td>
<td>References a specific generation of a data set</td>
</tr>
<tr>
<td></td>
<td>“INDEX= Data Set Option” on page 28</td>
<td>Defines indexes when a SAS data set is created</td>
</tr>
<tr>
<td></td>
<td>“LABEL= Data Set Option” on page 31</td>
<td>Specifies a label for the SAS data set</td>
</tr>
<tr>
<td></td>
<td>“OBSBUF= Data Set Option” on page 38</td>
<td>Determines the size of the view buffer for processing a DATA step view</td>
</tr>
<tr>
<td></td>
<td>“OUTREP= Data Set Option” on page 40</td>
<td>Specifies the data representation for the output SAS data set</td>
</tr>
<tr>
<td></td>
<td>“PW= Data Set Option” on page 43</td>
<td>Assigns a read, write, or alter password to a SAS file and enables access to a password-protected SAS file</td>
</tr>
<tr>
<td></td>
<td>“PWREQ= Data Set Option” on page 44</td>
<td>Controls the pop up of a requestor window for a data set password</td>
</tr>
<tr>
<td></td>
<td>“READ= Data Set Option” on page 44</td>
<td>Assigns a read password to a SAS file and enables access to a read-protected SAS file</td>
</tr>
<tr>
<td></td>
<td>“REPEMPTY= Data Set Option” on page 47</td>
<td>Controls replacement of like-named temporary or permanent SAS data sets when the new one is empty</td>
</tr>
<tr>
<td></td>
<td>“REPLACE= Data Set Option” on page 48</td>
<td>Controls replacement of like-named temporary or permanent SAS data sets</td>
</tr>
<tr>
<td></td>
<td>“REUSE= Data Set Option” on page 49</td>
<td>Specifies whether new observations are written to free space in compressed SAS data sets</td>
</tr>
<tr>
<td></td>
<td>“SORTEDBY= Data Set Option” on page 50</td>
<td>Specifies how the data set is currently sorted</td>
</tr>
<tr>
<td></td>
<td>“SORTSEQ= Data Set Option” on page 52</td>
<td>Specifies the collating sequence to be used by the SORT procedure</td>
</tr>
<tr>
<td></td>
<td>“SPILL= Data Set Option” on page 52</td>
<td>Specifies whether to create a spill file for non-sequential processing of a DATA step view</td>
</tr>
<tr>
<td></td>
<td>“TOBSNO= Data Set Option” on page 60</td>
<td>Specifies the number of observations to be transmitted in each multi-observation exchange with a SAS server</td>
</tr>
<tr>
<td></td>
<td>“TYPE= Data Set Option” on page 60</td>
<td>Specifies the data set type for a specially structured SAS data set</td>
</tr>
<tr>
<td></td>
<td>“WRITE= Data Set Option” on page 65</td>
<td>Assigns a write password to a SAS file and enables access to a write-protected SAS file</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>“FILECLOSE= Data Set Option” on page 20</td>
<td>Specifies how a tape is positioned when a SAS file on the tape is closed</td>
</tr>
<tr>
<td>Observation Control</td>
<td>“FIRSTOBS= Data Set Option” on page 20</td>
<td>Specifies which observation SAS processes first</td>
</tr>
</tbody>
</table>
### Dictionary

#### ALTER= Data Set Option

Assigns an alter password to a SAS file and enables access to a password-protected SAS file

**Valid in:** DATA step and PROC steps  
**Category:** Data Set Control  
**See:** ALTER= Data Set Option in the documentation for your operating environment.

**Syntax**

`ALTER=alter-password`
Syntax Description

*alter-password*

must be a valid SAS name. See “Rules for Words and Names in the SAS Language” in *SAS Language Reference: Concepts*.

Details

The ALTER= option applies to all types of SAS files except catalogs. You can use this option to assign an *alter-password* to a SAS file or to access a read-protected, write-protected, or alter-protected SAS file.

When replacing a SAS data set that is alter protected, the new data set inherits the alter password. To change the alter password for the new data set, use the MODIFY statement in the DATASETS procedure.

*Note:* A SAS password does not control access to a SAS file beyond the SAS system. You should use the operating system-supplied utilities and file-system security controls in order to control access to SAS files outside of SAS.

See Also

Data Set Options:
- “ENCRIPT= Data Set Option” on page 18
- “PW= Data Set Option” on page 43
- “READ= Data Set Option” on page 44
- “WRITE= Data Set Option” on page 65

“File Protection” in *SAS Language Reference: Concepts*

“Manipulating Passwords” in “The DATASETS Procedure” in *Base SAS Procedures Guide*

---

**BUFNO= Data Set Option**

Specifies the number of buffers to be allocated for processing a SAS data set

Valid in: DATA step and PROC steps

Category: Data Set Control

Syntax

BUFNO= n | nK | hexX | MIN | MAX
**Syntax Description**

\[ n \mid nK \]

specifies the number of buffers in multiples of 1 (bytes); 1,024 (kilobytes). For example, a value of 8 specifies 8 buffers, and a value of 1k specifies 1024 buffers.

**hex**

specifies the number of buffers as a hexadecimal value. You must specify the value beginning with a number (0-9), followed by an X. For example, the value 2dx sets the number of buffers to 45 buffers.

**MIN**

sets the minimum number of buffers to 0, which causes SAS to use the minimum optimal value for the operating environment. This is the default.

**MAX**

sets the number of buffers to the maximum possible number in your operating environment, up to the largest four-byte, signed integer, which is \(2^{31}-1\), or approximately 2 billion.

**Details**

The buffer number is not a permanent attribute of the data set; it is valid only for the current SAS session or job.

BUFFNO= applies to SAS data sets that are opened for input, output, or update.

A larger number of buffers can speed up execution time by limiting the number of input and output (I/O) operations that are required for a particular SAS data set. However, the improvement in execution time comes at the expense of increased memory consumption.

To reduce I/O operations on a small data set as well as speed execution time, allocate one buffer for each page of data to be processed. This technique is most effective if you read the same observations several times during processing.

**Comparisons**

- If the BUFFNO= data set option is not specified, then the value of the BUFFNO= system option is used. If both are specified in the same SAS session, the value specified for the BUFFNO= data set option overrides the value specified for the BUFFNO= system option.
- To request that SAS allocate the number of buffers based on the number of data set pages and index file pages, use the SASFILE global statement.

**See Also**

Data Set Options:

- “BUFSIZE= Data Set Option” on page 12

System Options:

- “BUFFNO= System Option” on page 1482

Statements:

- “SASFILE Statement” on page 1391
BUFSIZE= Data Set Option

Specifies the permanent buffer page size for an output SAS data set

Valid in: DATA step and PROC steps
Category: Data Set Control
Restriction: Use with output data sets only.
See: BUFSIZE= Data Set Option in the documentation for your operating environment.

Syntax

BUFSIZE= $n | nK | nM | nG | hexX | MAX$

Syntax Description

$n | nK | nM | nG$

specifies the page size in multiples of 1 (bytes); 1,024 (kilobytes); 1,048,576 (megabytes); or 1,073,741,824 (gigabytes). For example, a value of 8 specifies a page size of 8 bytes, and a value of 4k specifies a page size of 4096 bytes.

The default is 0, which causes SAS to use the minimum optimal page size for the operating environment.

$hexX$

specifies the page size as a hexadecimal value. You must specify the value beginning with a number (0-9), followed by an X. For example, the value 2dx sets the page size to 45 bytes.

MAX

sets the page size to the maximum possible number in your operating environment, up to the largest four-byte, signed integer, which is $n^{31}$-1, or approximately 2 billion bytes.

Details

The page size is the amount of data that can be transferred for a single I/O operation to one buffer. The page size is a permanent attribute of the data set and is used when the data set is processed.

A larger page size can speed up execution time by reducing the number of times SAS has to read from or write to the storage medium. However, the improvement in execution time comes at the cost of increased memory consumption.

To change the page size, use a DATA step to copy the data set and either specify a new page or use the SAS default. To reset the page size to the default value in your operating environment, use BUFSIZE=0.

Note: If you use the COPY procedure to copy a data set to another library that is allocated with a different engine, the specified page size of the data set is not retained.

Operating Environment Information: The default value for BUFSIZE= is determined by your operating environment and is set to optimize sequential access. To improve performance for direct (random) access, you should change the value for BUFSIZE=.
For the default setting and possible settings for direct access, see the BUFSIZE= data set option in the SAS documentation for your operating environment.

**Comparisons**

If the BUFSIZE= data set option is not specified, then the value of the BUFSIZE= system option is used. If both are specified in the same SAS session, the BUFSIZE= data set option overrides the value specified for the BUFSIZE= system option.

**See Also**

Data Set Options:
- “BUFNO= Data Set Option” on page 10

System Options:
- “BUFSIZE= System Option” on page 1483

---

**CNTLLEV= Data Set Option**

Specifies the level of shared access to a SAS data set

**Valid in:** DATA step and PROC steps

**Category:** Data Set Control

**Restriction:** Specify for input data sets only.

**Syntax**

CNTLLEV=LIB | MEM | REC

**Syntax Description**

**LIB**

specifies that concurrent access is controlled at the library level. Library-level control restricts concurrent access to only one update process to the library.

**MEM**

specifies that concurrent access is controlled at the SAS data set (member) level. Member-level control restricts concurrent access to only one update or output process to the SAS data set. If the data set is open for an update or output process, then no other operation can access the data set. If the data set is open for an input process, then other concurrent input processes are allowed but no update or output process is allowed.

**REC**

specifies that concurrent access is controlled at the observation (record) level. Record-level control allows more than one update access to the same SAS data set, but it denies concurrent update of the same observation.
Details

The CNTLLEV= option specifies the level at which shared update access to a SAS data set is denied. A SAS data set can be opened concurrently by more than one SAS session or by more than one statement, window, or procedure within a single session. By default, SAS procedures permit the greatest degree of concurrent access possible while they guarantee the integrity of the data and the data analysis. Therefore, you do not normally use the CNTLLEV= data set option.

Use this option when

- your application controls the access to the data, such as in SAS Component Language (SCL), SAS/IML software, or DATA step programming
- you access data through an interface engine that does not provide member-level control of the data.

If you use CNTLLEV=REC and the SAS procedure needs member-level control for integrity of the data analysis, SAS prints a warning to the SAS log that inaccurate or unpredictable results can occur if the data are updated by another process during the analysis.

Examples

Example 1: Changing the Shared Access Level

In the following example, the first SET statement includes the CNTLLEV= data set option in order to override the default level of shared access from member-level control to record-level control. The second SET statement opens the SAS data set with the default member-level control.

```sas
set datalib.fuel (cntllev=rec) point=obsnum;
.
.
set datalib.fuel;
by area;
```

COMPRESS= Data Set Option

Controls the compression of observations in an output SAS data set

Valid in: DATA step and PROC steps

Category: Data Set Control

Restriction: Use with output data sets only.

Syntax

COMPRESS=NO | YES | CHAR | BINARY

Syntax Description

NO
specifies that the observations in a newly created SAS data set are uncompressed (fixed-length records).

YES | CHAR
specifies that the observations in a newly created SAS data set are compressed (variable-length records) by SAS using RLE (Run Length Encoding). RLE compresses observations by reducing repeated consecutive characters (including blanks) to two-byte or three-byte representations.

Alias: ON
Tip: Use this compression algorithm for character data.

Note: COMPRESS=CHAR is accepted by Version 7 and later versions.

BINARY
specifies that the observations in a newly created SAS data set are compressed (variable-length records) by SAS using RDC (Ross Data Compression). RDC combines run-length encoding and sliding-window compression to compress the file.

Tip: This method is highly effective for compressing medium to large (several hundred bytes or larger) blocks of binary data (numeric variables). Because the compression function operates on a single record at a time, the record length needs to be several hundred bytes or larger for effective compression.

Details
Compressing a file is a process that reduces the number of bytes required to represent each observation. Advantages of compressing a file include reduced storage requirements for the file and fewer I/O operations necessary to read or write to the data during processing. However, more CPU resources are required to read a compressed file (because of the overhead of uncompressing each observation), and there are situations where the resulting file size might increase rather than decrease.

Use the COMPRESS= data set option to compress an individual file. Specify the option for output data sets only—that is, data sets named in the DATA statement of a DATA step or in the OUT= option of a SAS procedure. Use the COMPRESS= data set option only when you are creating a SAS data file (member type DATA). You cannot compress SAS views, because they contain no data.

After a file is compressed, the setting is a permanent attribute of the file, which means that to change the setting, you must re-create the file. That is, to uncompress a file, specify COMPRESS=NO for a DATA step that copies the compressed file.

Comparisons
The COMPRESS= data set option overrides the COMPRESS= option on the LIBNAME statement and the COMPRESS= system option.

The data set option POINTOBS=YES, which is the default, determines that a compressed data set can be processed with random access (by observation number) rather than sequential access. With random access, you can specify an observation number in the FSEDIT procedure and the POINT= option in the SET and MODIFY statements.

When you create a compressed file, you can also specify REUSE=YES (as a data set option or system option) in order to track and reuse space. With REUSE=YES, new observations are inserted in space freed when other observations are updated or deleted. When the default REUSE=NO is in effect, new observations are appended to the existing file.

POINTOBS=YES and REUSE=YES are mutually exclusive—that is, they cannot be used together. REUSE=YES takes precedence over POINTOBS=YES; that is, if you set REUSE=YES, SAS automatically sets POINTOBS=NO. For example, the following
statement results in a compressed data file that cannot be processed with random access:

  The TAPE engine supports the COMPRESS= data set option, but the engine does not support the COMPRESS= system option.
  The XPORT engine does not support compression.

See Also

Data Set Options:
  “POINTOBS= Data Set Option” on page 42
  “REUSE= Data Set Option” on page 49
Statements:
  “LIBNAME Statement” on page 1286
System Options:
  “COMPRESS= System Option” on page 1500
  “REUSE= System Option” on page 1591
  “Compressing Data Files” in SAS Language Reference: Concepts

DLDMGACTION= Data Set Option

Specifies what type of action to take when a SAS data set in a SAS data library is detected as damaged

Valid in:  DATA step and PROC steps
Category:  Data Set Control

Syntax

DLDMGACTION=FAIL | ABORT | REPAIR | PROMPT

Syntax Description

FAIL
  stops the step, issues an error message to the log immediately. This is the default for batch mode.

ABORT
  terminates the step, issues an error message to the log, and aborts the SAS session.

REPAIR
  automatically repairs and rebuilds indexes and integrity constraints, unless the data set is truncated. You use the REPAIR statement in PROC DATASETS to restore a truncated data set. It issues a warning message to the log. This is the default for interactive mode.

PROMPT
  displays a requestor window that asks you to select the FAIL, ABORT, or REPAIR action.
**DROP= Data Set Option**

Excludes variables from processing or from output SAS data sets

Valid in: DATA step and PROC steps  
Category: Variable Control

**Syntax**

DROP=variable(s)

**Syntax Description**

*variable(s)*

lists one or more variable names. You can list the variables in any form that SAS allows.

**Details**

If the option is associated with an input data set, the variables are not available for processing. If the DROP= data set option is associated with an output data set, SAS does not write the variables to the output data set, but they are available for processing.

**Comparisons**

- The DROP= data set option differs from the DROP statement in these ways:
  - In DATA steps, the DROP= data set option can apply to both input and output data sets. The DROP statement applies only to output data sets.
  - In DATA steps, when you create multiple output data sets, use the DROP= data set option to write different variables to different data sets. The DROP statement applies to all output data sets.
  - In PROC steps, you can use only the DROP= data set option, not the DROP statement.
  - The KEEP= data set option specifies a list of variables to be included in processing or to be written to the output data set.

**Examples**

**Example 1: Excluding Variables from Input**  
In this example, the variables SALARY and GENDER are not included in processing and they are not written to either output data set:

```sas
data plan1 plan2;
  set payroll(drop=salary gender);
  if hired<’01jan98’d then output plan1;
  else output plan2;
run;
```
You cannot use SALARY or GENDER in any logic in the DATA step because DROP= prevents the SET statement from reading them from PAYROLL.

**Example 2: Processing Variables without Writing Them to a Data Set**  
In this example, SALARY and GENDER are not written to PLAN2, but they are written to PLAN1:

```sas
data plan1 plan2(drop=salary gender);
  set payroll;
  if hired<’01jan98’d then output plan1;
  else output plan2;
run;
```

**See Also**

Data Set Options:

“KEEP= Data Set Option” on page 30  
Statements:  
“DROP Statement” on page 1152

---

**ENCODING= Data Set Option**

Overrides the encoding to use for reading or writing a SAS data set  
Valid in: DATA step and PROC steps  
Category: Data Set Control  

---

**ENCRYPT= Data Set Option**

Encrypts SAS data files  
Valid in: DATA step and PROC steps  
Category: Data Set Control  
Restriction: Use with output data sets only.

**Syntax**

ENCRYPT=YES | NO

**Syntax Description**

YES
encrypts the file. The encryption method uses passwords. At a minimum, you must specify the READ= or the PW= data set option at the same time that you specify ENCRYPT=YES. Because the encryption method uses passwords, you cannot change any password on an encrypted data set without re-creating the data set.

NO
does not encrypt the file.

CAUTION:
Record all passwords. If you forget the password, you cannot reset it without assistance from SAS. The process is time-consuming and resource-intensive.

Details

- You can use the ENCRYPT= option only when you are creating a SAS data file.
- In order to copy an encrypted SAS data file, the output engine must support encryption. Otherwise, the data file is not copied.
- Encrypted files work only in Release 6.11 or in later releases of SAS.
- You cannot encrypt SAS data views or stored programs because they contain no data.
- If the data file is encrypted, all associated indexes are also encrypted.
- Encryption requires roughly the same amount of CPU resources as compression.
- You cannot use PROC CPORT on encrypted SAS data files.

Example

This example creates an encrypted SAS data set:

data salary(encrypt=yes read=green);
  input name $ yrsal bonuspct;
  datalines;
  Muriel  34567  3.2
  Bjorn   74644  2.5
  Freda   38755  4.1
  Benny   29855  3.5
  Agnetha 70998  4.1
; 

to use this data set, specify the read password:

proc contents data=salary(read=green);
run;

See Also

Data Set Options:

“ALTER= Data Set Option” on page 9
“PW= Data Set Option” on page 43
“READ= Data Set Option” on page 44
“WRITE= Data Set Option” on page 65
“SAS Data File Encryption” in SAS Language Reference: Concepts
**FILECLOSE= Data Set Option**

Specifies how a tape is positioned when a SAS file on the tape is closed

**Valid in:** DATA step and PROC steps

**Category:** Miscellaneous

**Syntax**

FILECLOSE=DISP | LEAVE | REREAD | REWIND

**Syntax Description**

**DISP**
positions the tape volume according to the disposition specified in the operating environment's control language.

**LEAVE**
positions the tape at the end of the file that was just processed. Use FILECLOSE=LEAVE if you are not repeatedly accessing the same files in a SAS program but you are accessing one or more subsequent SAS files on the same tape.

**REREAD**
positions the tape volume at the beginning of the file that was just processed. Use FILECLOSE=REREAD if you are accessing the same SAS data set on tape several times in a SAS program.

**REWIND**
rewinds the tape volume to the beginning. Use FILECLOSE=REWIND if you are accessing one or more previous SAS files on the same tape, but you are not repeatedly accessing the same files in a SAS program.

*Operating Environment Information:* These values are not recognized by all operating environments. Additional values are available on some operating environments. See the appropriate sections of the SAS documentation for your operating environment for more information on using SAS data libraries that are stored on tape.

**FIRSTOBS= Data Set Option**

Specifies which observation SAS processes first

**Valid in:** DATA step and PROC steps

**Category:** Observation Control

**Restriction:** Valid for input (read) processing only.

**Restriction:** Cannot use with PROC SQL views.

**Syntax**

FIRSTOBS= n | nK | nM | nG | hexX | MIN | MAX
Syntax Description

\[ n \mid nK \mid nM \mid nG \]
specifies the number of the first observation to process in multiples of 1 (bytes); 1,024 (kilobytes); 1,048,576 (megabytes); or 1,073,741,824 (gigabytes). For example, a value of \textit{8} specifies the 8th observation, and a value of \textit{3k} specifies 3,072.

\[ \text{hexX} \]
specifies the number of the first observation to process as a hexadecimal value. You must specify the value beginning with a number (0-9), followed by an X. For example, the value \textit{2dx} sets the 45th observation as the first observation to process.

\textbf{MIN}
sets the number of the first observation to process to 1. This is the default.

\textbf{MAX}
sets the number of the first observation to process to the maximum number of observations in the data set, up to the largest eight-byte, signed integer, which is \(2^{63}-1\), or approximately 9.2 quintillion observations.

Details

The FIRSTOBS= data set option affects a single, existing SAS data set. Use the FIRSTOBS= system option to affect all steps for the duration of your current SAS session. FIRSTOBS= is valid for input (read) processing only. Specifying FIRSTOBS= is not valid for output or update processing.

You can apply FIRSTOBS= processing to WHERE processing. For more information, see “Processing a Segment of Data That Is Conditionally Selected” in SAS Language Reference: Concepts.

Comparisons

- The FIRSTOBS= data set option overrides the FIRSTOBS= system option for the individual data set.
- While the FIRSTOBS= data set option specifies a starting point for processing, the OBS= data set option specifies an ending point. The two options are often used together to define a range of observations to be processed.
- When external files are read, the FIRSTOBS= option in the INFILE statement specifies which record to read first.

Examples

This PROC step prints the data set STUDY beginning with observation 20:

```sas
proc print data=study(firstobs=20);
run;
```

This SET statement uses both FIRSTOBS= and OBS= to read only observations 5 through 10 from the data set STUDY. Data set NEW contains six observations.

```sas
data new;
set study(firstobs=5 obs=10);
run;
```
See Also

Data Set Options:
   “OBS= Data Set Option” on page 32
Statements:
   “INFILE Statement” on page 1224
   “WHERE Statement” on page 1421
System Options:
   “FIRSTOBS= System Option” on page 1529

GENMAX= Data Set Option

Requests generations for a data set and specifies the maximum number of versions

Valid in:    DATA step and PROC steps
Category:   Data Set Control
Restriction: Use with output data sets only.

Syntax

GENMAX=number-of-generations

Syntax Description

number-of-generations

requests generations for a data set and specifies the maximum number of versions to
maintain. The value can be from 0 to 1000. The default is GENMAX=0, which means
that no generation data sets are requested.

Details

You use GENMAX= to request generations for a new data set and to modify the number
of generations on an existing data set. The first time the data set is replaced, SAS
keeps the replaced version and appends a four-character version number to its member
name, which includes # and a three-digit number. For example, for a data set named A,
a historical version would be A#001.

Once generations of a data set is requested, its member name is limited to 28
characters (rather than 32), because the last four characters are reserved for the
appended version number. When the GENMAX= data set option is set to 0, the member
name can be up to 32 characters.

If you reduce the number of generations on an existing data set, SAS deletes the
oldest version(s) above the new limit.

Examples

Example 1: Requesting Generations When You Create a Data Set   This example shows
how to request generations for a new data set. The DATA step creates a data set named
WORK.A that can have as many as 10 generations (one current version and nine historical versions):

```sas
data a(genmax=10);
  x=1;
  output;
run;
```

**Example 2: Modifying the Number of Generations on an Existing Data Set**

This example shows how to change the number of generations on the data set MYLIB.A to 4:

```sas
proc datasets lib=mylib;
  modify a(genmax=4);
run;
```

**See Also**

Data Set Option:

“GENNUM= Data Set Option” on page 23

“Generation Data Sets” in “SAS Data Sets” in *SAS Language Reference: Concepts*
For the CHANGE statement, GENNUM= supports the additional value ALL.

For the CHANGE statement, specifying GENNUM=0 refers to all versions rather than just the base version.

Details

After generations for a data set have been requested using the GENMAX= data set option, use GENNUM= to request a specific version. For example, specifying GENNUM=3 refers to the historical version #003, while specifying GENNUM=-1 refers to the youngest historical version.

Note that after 999 replacements, the youngest version would be #999. After 1,000 replacements, SAS rolls over the youngest version number to #000. Therefore, if you want the historical version #000, specify GENNUM=1000.

Both an absolute reference and a relative reference refer to a specific version. A relative reference does not skip deleted versions. Therefore, when working with a generation group that includes one or more deleted versions, using a relative reference will result in an error if the version being referenced has been deleted. For example, if you have the base version AIR and three historical versions (AIR#001, AIR#002, and AIR#003) and you delete AIR#002, the following statements return an error, because AIR#002 does not exist. SAS does not assume you mean AIR#003:

```sas
proc print data=air (gennum= -2);
run;
```

Examples

Example 1: Requesting a Version Using an Absolute Reference

This example prints the historical version #003 for data set A, using an absolute reference:

```sas
proc print data=a(gennum=3);
run;
```

Example 2: Requesting A Version Using a Relative Reference

The following PRINT procedure prints the data set three versions back from the base version:

```sas
proc print data=a(gennum=-3);
run;
```

See Also

Data Set Option:

“GENMAX= Data Set Option” on page 22

“Understanding Generation Data Sets” in “SAS Data Files” in SAS Language Reference: Concepts

“The DATASETS Procedure” in the Base SAS Procedures Guide
IDXNAME= Data Set Option

Directs SAS to use a specific index to satisfy the conditions of a WHERE expression

Valid in: DATA step and PROC steps
Category: User Control of SAS Index Usage
Restriction: Use with input data sets only
Restriction: Mutually exclusive with IDXWHERE= data set option

Syntax
IDXNAME=index-name

Syntax Description

index-name specifies the name (up to 32 characters) of a simple or composite index for the SAS data set. SAS does not attempt to determine if the specified index is the best one or if a sequential search might be more resource efficient.

Interaction: The specification is not a permanent attribute of the data set and is valid only for the current use of the data set.

Tip: To request that IDXNAME= usage be noted in the SAS log, specify the system option MSGLEVEL=I.

Details

By default, to satisfy the conditions of a WHERE expression for an indexed SAS data set, SAS identifies zero or more candidate indexes that could be used to optimize the WHERE expression. From the list of candidate indexes, SAS selects the one that it determines will provide the best performance, or rejects all of the indexes if a sequential pass of the data is expected to be more efficient.

Because the index SAS selects might not always provide the best optimization, you can direct SAS to use one of the candidate indexes by specifying the IDXNAME= data set option. If you specify an index that SAS does not identify as a candidate index, then IDXNAME= will not process the request; that is, IDXNAME= will not allow you to specify an index that would produce incorrect results.

Comparisons

IDXWHERE= enables you to override the SAS decision about whether to use an index.

Example

This example uses the IDXNAME= data set option in order to direct SAS to use a specific index to optimize the WHERE expression. SAS then disregards the possibility that a sequential search of the data set might be more resource efficient and does not attempt to determine if the specified index is the best one. (Note that the EMPNUM index was not created with the NOMISS option.)
data mydata.empnew;
  set mydata.employee (idxname=empnum);
  where empnum < 2000;
run;

See Also

Data Set Option:
“IDXWHERE= Data Set Option” on page 26
“Using an Index for WHERE Processing” in SAS Language Reference: Concepts
“WHERE-Expression Processing” in SAS Language Reference: Concepts

IDXWHERE= Data Set Option

Overriding the SAS decision about whether to use an index to satisfy the conditions of a WHERE expression

Valid in: DATA step and PROC steps
Category: User Control of SAS Index Usage
Restriction: Use with input data sets only.
Restriction: Mutually exclusive with IDXNAME= data set option

Syntax
IDXWHERE= YES | NO

Syntax Description

YES
tells SAS to choose the best index to optimize a WHERE expression, and to disregard the possibility that a sequential search of the data set might be more resource-efficient.

NO
tells SAS to ignore all indexes and satisfy the conditions of a WHERE expression with a sequential search of the data set.

Note: You cannot use IDXWHERE= to override the use of an index to process a BY statement.

Details

By default, to satisfy the conditions of a WHERE expression for an indexed SAS data set, SAS decides whether to use an index or to read the data set sequentially. The software estimates the relative efficiency and chooses the method that is more efficient. You might need to override the software's decision by specifying the IDXWHERE= data set option because the decision is based on general rules that may occasionally not
produce the best results. That is, by specifying the IDXWHERE= data set option, you are able to determine the processing method.

*Note:* The specification is not a permanent attribute of the data set and is valid only for the current use of the data set.

*Note:* If you issue the system option MSGLEVEL=I, you can request that IDXWHERE= usage be noted in the SAS log if the setting affects index processing.

Comparisons
IDXNAME= enables you to direct SAS to use a specific index.

Examples

**Example 1: Specifying Index Usage**  This example uses the IDXWHERE= data set option to tell SAS to decide which index is the best to optimize the WHERE expression. SAS then disregards the possibility that a sequential search of the data set might be more resource-efficient:

```sas
data mydata.empnew;
  set mydata.employee (idxwhere=yes);
  where empnum < 2000;
```

**Example 2: Specifying No Index Usage**  This examples uses the IDXWHERE= data set option to tell SAS to ignore any index and to satisfy the conditions of the WHERE expression with a sequential search of the data set:

```sas
data mydata.empnew;
  set mydata.employee (idxwhere=no);
  where empnum < 2000;
```

See Also

Data Set Option:

“IDXNAME= Data Set Option” on page 25

“Understanding SAS Indexes” in the “SAS Data Files” section in SAS Language Reference: Concepts

“WHERE-Expression Processing” in SAS Language Reference: Concepts

---

**IN= Data Set Option**

Creates a variable that indicates whether the data set contributed data to the current observation

**Valid in:** DATA step

**Category:** Observation Control

**Restriction:** Use with the SET, MERGE, MODIFY, and UPDATE statements only.
Syntax
IN=variable

Syntax Description

variable
names the new variable whose value indicates whether that input data set contributed data to the current observation. Within the DATA step, the value of the variable is 1 if the data set contributed to the current observation, and 0 otherwise.

Details
Specify the IN= data set option in parentheses after a SAS data set name in the SET, MERGE, MODIFY and UPDATE statements only. Values of IN= variables are available to program statements during the DATA step, but the variables are not included in the SAS data set that is being created, unless they are explicitly assigned to a new variable.

When you use IN= with BY-group processing, and when a data set contributes an observation for the current BY group, the IN= value is 1. The value remains as long as that BY group is still being processed and the value is not reset by programming logic.

Examples

In this example, IN= creates a new variable, OVERSEAS, that denotes international flights. The variable I has a value of 1 when the observation is read from the NONUSA data set; otherwise, it has a value of 0. The IF-THEN statement checks the value of I to determine if the data set NONUSA contributed data to the current observation. If I=1, the variable OVERSEAS receives an asterisk (*) as a value.

```sas
data allflts;
  set usa nonusa(in=i);
  by fltnum;
  if i then overseas='*';
run;
```

See Also

Statements:
“BY Statement” on page 1114
“MERGE Statement” on page 1306
“MODIFY Statement” on page 1310
“SET Statement” on page 1399
“UPDATE Statement” on page 1416
“BY-Group Processing” in SAS Language Reference: Concepts

INDEX= Data Set Option

Defines indexes when a SAS data set is created
Valid in: DATA step and PROC steps
Category: Data Set Control
Restriction: Use with output data sets only.

Syntax
INDEX=(index-specification-1 ...<index-specification-n>)

Syntax Description

**index-specification**
names and describes a simple or a composite index to be built. *Index-specification* has this form:

```
index <= (variable(s)) > </UNIQUE> </NOMISS>
```

- **index** is the name of a variable that forms the index or the name you choose for a composite index.
- **variable(s)** is a list of variables to use in making a composite index.
- **UNIQUE** specifies that the values of the key variables must be unique. If you specify UNIQUE for a new data set and multiple observations have the same values for the index variables, the index is not created. A slash (/) must precede the UNIQUE option.
- **NOMISS** excludes all observations with missing values from the index. Observations with missing values are still read from the data set but not through the index. A slash (/) must precede the NOMISS option.

Examples

**Example 1: Defining a Simple Index**  The following INDEX= data set option defines a simple index for the SSN variable:

```
data new(index=(ssn));
```

**Example 2: Defining a Composite Index**  The following INDEX= data set option defines a composite index named CITYST that uses the CITY and STATE variables:

```
data new(index=(cityst=(city state)));
```

**Example 3: Defining a Simple and a Composite Index**  The following INDEX= data set option defines a simple index for SSN and a composite index for CITY and STATE:

```
data new(index=(ssn cityst=(city state)));
```

See Also

INDEX CREATE statement in “The DATASETS Procedure” in *Base SAS Procedures Guide*
CREATE INDEX statement in “The SQL Procedure” in Base SAS Procedures Guide
“Understanding SAS Indexes” in the “SAS Data Files” section of SAS Language Reference: Concepts

**KEEP= Data Set Option**

Specifies variables for processing or for writing to output SAS data sets

Valid in: DATA step and PROC steps

Category: Variable Control

**Syntax**

KEEP=variable(s)

**Syntax Description**

variable(s) lists one or more variable names. You can list the variables in any form that SAS allows.

**Details**

If the KEEP= data set option is associated with an input data set, only those variables that are listed after the KEEP= data set option are available for processing. If the KEEP= data set option is associated with an output data set, only the variables listed after the option are written to the output data set, but all variables are available for processing.

**Comparisons**

- The KEEP= data set option differs from the KEEP statement in the following ways:
  - In DATA steps, the KEEP= data set option can apply to both input and output data sets. The KEEP statement applies only to output data sets.
  - In DATA steps, when you create multiple output data sets, use the KEEP= data set option to write different variables to different data sets. The KEEP statement applies to all output data sets.
  - In PROC steps, you can use only the KEEP= data set option, not the KEEP statement.
  - The DROP= data set option specifies variables to omit during processing or to omit from the output data set.
Example

In this example, only IDNUM and SALARY are read from PAYROLL, and they are the only variables in PAYROLL that are available for processing:

```sas
data bonus;
  set payroll(keep=idnum salary);
  bonus=salary*1.1;
run;
```

See Also

Data Set Options:
  “DROP= Data Set Option” on page 17

Statements:
  “KEEP Statement” on page 1278

---

**LABEL= Data Set Option**

Specifies a label for the SAS data set

Valid in: DATA step and PROC steps

Category: Data Set Control

---

**Syntax**

LABEL='label'

**Syntax Description**

'label'

is a text string of up to 256 characters. If the label text contains single quotation marks, use double quotation marks around the label, or use two single quotation marks in the label text and surround the string with single quotation marks. To remove a label from a data set, assign a label that is equal to a blank that is enclosed in quotation marks.

**Details**

You can use the LABEL= option on both input and output data sets. When you use LABEL= on input data sets, it assigns a label for the file for the duration of that DATA or PROC step. When it is specified for an output data set, the label becomes a permanent part of that file and can be printed using the CONTENTS or DATASETS procedure, and modified using PROC DATASETS.

A label assigned to a data set remains associated with that data set when you update a data set in place, such as when you use the APPEND procedure or the MODIFY
However, a label is lost if you use a data set with a previously assigned label to create a new data set in the DATA step. For example, a label previously assigned to data set ONE is lost when you create the new output data set ONE in this DATA step:

```sas
data one;
  set one;
run;
```

### Comparisons

- The LABEL= data set option enables you to specify labels only for data sets. You can specify labels for the variables in a data set using the LABEL statement.
- The LABEL= option in the ATTRIB statement also enables you to assign labels to variables.

### Examples

These examples assign labels to data sets:

```sas
data w2(label='1976 W2 Info, Hourly');

data new(label='Peter''s List');

data new(label="Hillside's Daily Account");

data sales(label='Sales For May(NE)');
```

### See Also

Statements:

- “ATTRIB Statement” on page 1111
- “LABEL Statement” on page 1280
- “MODIFY Statement” on page 1310
- “The CONTENTS Procedure” in Base SAS Procedures Guide
- “The DATASETS Procedure” in Base SAS Procedures Guide

---

**OBS= Data Set Option**

Specifies when to stop processing observations

**Valid in:** DATA step and PROC steps

**Category:** Observation Control

**Restriction:** Use with input data sets only

**Restriction:** Cannot use with PROC SQL views

**Default:** MAX
SAS Data Set Options  △  OBS= Data Set Option  33

---

**Syntax**

OBS= $n \mid nK \mid nM \mid nG \mid nT \mid hexX \mid MIN \mid MAX$

**Syntax Description**

$n \mid nK \mid nM \mid nG \mid nT$

specifies a number to indicate when to stop processing observations, with $n$ being an integer. Using one of the letter notations results in multiplying the integer by a specific value. That is, specifying K (kilo) multiplies the integer by 1,024, M (mega) multiplies by 1,048,576, G (giga) multiplies by 1,073,741,824, or T (tera) multiplies by 1,099,511,627,776. For example, a value of 20 specifies 20 observations, while a value of 3m specifies 3,145,728 observations.

$hexX$

specifies a number to indicate when to stop processing as a hexadecimal value. You must specify the value beginning with a number (0–9), followed by an X. For example, the hexadecimal value F8 must be specified as 0F8X in order to specify the decimal equivalent of 248. The value 2dx specifies the decimal equivalent of 45.

**MIN**

sets the number to indicate when to stop processing to 0. Use OBS=0 in order to create an empty data set that has the structure, but not the observations, of another data set.

**Interaction:** If OBS=0 and the NOREPLACE option is in effect, then SAS can still take certain actions because it actually executes each DATA and PROC step in the program, using no observations. For example, SAS executes procedures, such as CONTENTS and DATASETS, that process libraries or SAS data sets.

**MAX**

sets the number to indicate when to stop processing to the maximum number of observations in the data set, up to the largest 8-byte, signed integer, which is $2^{31}$-1, or approximately 9.2 quintillion. This is the default.

**Details**

OBS= tells SAS when to stop processing observations. To determine when to stop processing, SAS uses the value for OBS= in a formula that includes the value for OBS= and the value for FIRSTOBS=. The formula is

$$(\text{obs} - \text{firstobs}) + 1 = \text{results}$$

For example, if OBS=10 and FIRSTOBS=1 (which is the default for FIRSTOBS=), the result is ten observations, that is, $(10 - 1) + 1 = 10$. If OBS=10 and FIRSTOBS=2, the result is nine observations, that is, $(10 - 2) + 1 = 9$. OBS= is valid only when an existing SAS data set is read.

**Comparisons**

- The OBS= data set option overrides the OBS= system option for the individual data set.
While the OBS= data set option specifies an ending point for processing, the FIRSTOBS= data set option specifies a starting point. The two options are often used together to define a range of observations to be processed.

The OBS= data set option enables you to select observations from SAS data sets. You can select observations to be read from external data files by using the OBS= option in the INFILE statement.

Examples

Example 1: Using OBS= to Specify When to Stop Processing Observations

This example illustrates the result of using OBS= to tell SAS when to stop processing observations. This example creates a SAS data set, then executes the PRINT procedure with FIRSTOBS=2 and OBS=12. The result is 11 observations, that is, \((12 - 2) + 1 = 11\). The result of OBS= in this situation appears to be the observation number that SAS processes last, because the output starts with observation 2, and ends with observation 12, but this is only a coincidence.

```sas
data Ages;
  input Name $ Age;
  datalines;
  Miguel 53
  Brad 27
  Willie 69
  Marc 50
  Sylvia 40
  Arun 25
  Gary 40
  Becky 51
  Alma 39
  Tom 62
  Kris 66
  Paul 60
  Randy 43
  Barbara 52
  Virginia 72;
  proc print data=Ages (firstobs=2 obs=12);
  run;
```
Output 2.1  PROC PRINT Output Using OBS= and FIRSTOBS=

The SAS System

<table>
<thead>
<tr>
<th>Obs</th>
<th>Name</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Brad</td>
<td>27</td>
</tr>
<tr>
<td>3</td>
<td>Willie</td>
<td>69</td>
</tr>
<tr>
<td>4</td>
<td>Marc</td>
<td>50</td>
</tr>
<tr>
<td>5</td>
<td>Sylvia</td>
<td>40</td>
</tr>
<tr>
<td>6</td>
<td>Arun</td>
<td>25</td>
</tr>
<tr>
<td>7</td>
<td>Gary</td>
<td>40</td>
</tr>
<tr>
<td>8</td>
<td>Becky</td>
<td>51</td>
</tr>
<tr>
<td>9</td>
<td>Alma</td>
<td>39</td>
</tr>
<tr>
<td>10</td>
<td>Tom</td>
<td>62</td>
</tr>
<tr>
<td>11</td>
<td>Kris</td>
<td>66</td>
</tr>
<tr>
<td>12</td>
<td>Paul</td>
<td>60</td>
</tr>
</tbody>
</table>

Example 2: Using OBS= with WHERE Processing  This example illustrates the result of using OBS= along with WHERE processing. The example uses the data set that was created in Example 1, which contains 15 observations.

First, here is the PRINT procedure with a WHERE statement. The subset of the data results in 12 observations:

```
proc print data=Ages;
  where Age LT 65;
run;
```

Output 2.2  PROC PRINT Output Using a WHERE Statement

The SAS System

<table>
<thead>
<tr>
<th>Obs</th>
<th>Name</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Miguel</td>
<td>53</td>
</tr>
<tr>
<td>2</td>
<td>Brad</td>
<td>27</td>
</tr>
<tr>
<td>4</td>
<td>Marc</td>
<td>50</td>
</tr>
<tr>
<td>5</td>
<td>Sylvia</td>
<td>40</td>
</tr>
<tr>
<td>6</td>
<td>Arun</td>
<td>25</td>
</tr>
<tr>
<td>7</td>
<td>Gary</td>
<td>40</td>
</tr>
<tr>
<td>8</td>
<td>Becky</td>
<td>51</td>
</tr>
<tr>
<td>9</td>
<td>Alma</td>
<td>39</td>
</tr>
<tr>
<td>10</td>
<td>Tom</td>
<td>62</td>
</tr>
<tr>
<td>12</td>
<td>Paul</td>
<td>60</td>
</tr>
</tbody>
</table>

Executing the PRINT procedure with the WHERE statement and OBS=10 results in 10 observations, that is, \((10 - 1) + 1 = 10\). Note that with WHERE processing, SAS first subsets the data, then applies OBS= to the subset.

```
proc print data=Ages (obs=10);
  where Age LT 65;
run;
```
Output 2.3  PROC PRINT Output Using a WHERE Statement and OBS=

<table>
<thead>
<tr>
<th>The SAS System</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obs</td>
<td>Name</td>
</tr>
<tr>
<td>1</td>
<td>Miguel</td>
</tr>
<tr>
<td>2</td>
<td>Brad</td>
</tr>
<tr>
<td>4</td>
<td>Marc</td>
</tr>
<tr>
<td>5</td>
<td>Sylvia</td>
</tr>
<tr>
<td>6</td>
<td>Arun</td>
</tr>
<tr>
<td>7</td>
<td>Gary</td>
</tr>
<tr>
<td>8</td>
<td>Becky</td>
</tr>
<tr>
<td>9</td>
<td>Alma</td>
</tr>
<tr>
<td>10</td>
<td>Tom</td>
</tr>
<tr>
<td>12</td>
<td>Paul</td>
</tr>
</tbody>
</table>

The result of OBS= appears to be how many observations to process, because the output consists of 10 observations, ending with the observation number 12. However, the result is only a coincidence. If you apply FIRSTOBS=2 and OBS=10 to the subset, then the result is nine observations, that is, \((10 - 2) + 1 = 9\). OBS= in this situation is neither the observation number to end with nor how many observations to process; the value is used in the formula to determine when to stop processing.

```
proc print data=Ages (firstobs=2 obs=10);
  where Age LT 65;
run;
```

Output 2.4  PROC PRINT Output Using WHERE Statement, OBS=, and FIRSTOBS=

<table>
<thead>
<tr>
<th>The SAS System</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obs</td>
<td>Name</td>
</tr>
<tr>
<td>2</td>
<td>Brad</td>
</tr>
<tr>
<td>4</td>
<td>Marc</td>
</tr>
<tr>
<td>5</td>
<td>Sylvia</td>
</tr>
<tr>
<td>6</td>
<td>Arun</td>
</tr>
<tr>
<td>7</td>
<td>Gary</td>
</tr>
<tr>
<td>8</td>
<td>Becky</td>
</tr>
<tr>
<td>9</td>
<td>Alma</td>
</tr>
<tr>
<td>10</td>
<td>Tom</td>
</tr>
<tr>
<td>12</td>
<td>Paul</td>
</tr>
</tbody>
</table>

Example 3: Using OBS= When Observations Are Deleted  This example illustrates the result of using OBS= for a data set that has deleted observations. The example uses the data set that was created in Example 1, with observation 6 deleted.

First, here is PROC PRINT output of the modified file:

```
proc print data=Ages;
run;
```
Executing the PRINT procedure with OBS=12 results in 12 observations, that is, \((12 - 1) + 1 = 12\):

```
proc print data=Ages (obs=12);
run;
```

The result of OBS= appears to be how many observations to process, because the output consists of 12 observations, ending with the observation number 13. However, if you apply FIRSTOBS=2 and OBS=12, the result is 11 observations, that is, \((12 - 2) + 1 = 11\). OBS= in this situation is neither the observation number to end with nor how many observations to process; the value is used in the formula to determine when to stop processing.

```
proc print data=Ages (firstobs=2 obs=12);
run;
```
Output 2.7  PROC PRINT Output Using OBS= and FIRSTOBS=

<table>
<thead>
<tr>
<th>Obs</th>
<th>Name</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Brad</td>
<td>27</td>
</tr>
<tr>
<td>3</td>
<td>Willie</td>
<td>69</td>
</tr>
<tr>
<td>4</td>
<td>Marc</td>
<td>50</td>
</tr>
<tr>
<td>5</td>
<td>Sylvia</td>
<td>40</td>
</tr>
<tr>
<td>7</td>
<td>Gary</td>
<td>40</td>
</tr>
<tr>
<td>8</td>
<td>Becky</td>
<td>51</td>
</tr>
<tr>
<td>9</td>
<td>Alma</td>
<td>39</td>
</tr>
<tr>
<td>10</td>
<td>Tom</td>
<td>62</td>
</tr>
<tr>
<td>11</td>
<td>Kris</td>
<td>66</td>
</tr>
<tr>
<td>12</td>
<td>Paul</td>
<td>60</td>
</tr>
<tr>
<td>13</td>
<td>Randy</td>
<td>43</td>
</tr>
</tbody>
</table>

See Also

Data Set Options:
“FIRSTOBS= Data Set Option” on page 20

Statements:
“INFILE Statement” on page 1224
“WHERE Statement” on page 1421

System Options:
“OBS= System Option” on page 1572

For more information about using OBS= with WHERE processing, see “Processing a Segment of Data That Is Conditionally Selected” in SAS Language Reference: Concepts.

OBSBUF= Data Set Option

Determines the size of the view buffer for processing a DATA step view

Valid in: DATA step and PROC steps
Category: Data Set Control
Restriction: Valid only for a DATA step view

Syntax
OBSBUF=n

Syntax Description

n specifies the number of observations that are read into the view buffer at a time.
Default: 32K bytes of memory are allocated for the default view buffer, which means that the default number of observations that can be read into the view buffer at one time depends on the observation length. Therefore, the default is the number of observations that can fit into 32K bytes. If the observation length is larger than 32K, then only one observation can be read into the buffer at a time.

Tip: To determine the observation length, which is its size in bytes, use PROC CONTENTS for the DATA step view.

Details

The OBSBUF= data set option specifies the number of observations that can be read into the view buffer at a time. The view buffer is a segment of memory that is allocated to hold output observations that are generated from a DATA step view. The size of the buffer determines how much data can be held in memory at one time. OBSBUF= enables you to tune the performance of reading data from a DATA step view.

The view buffer is shared between the request that opens the DATA step view, for example, a SAS procedure, and the DATA step view itself. Two computer tasks coordinate between requesting data and generating and returning the data as follows:

1. When a request task, such as a PRINT procedure, requests data, task switching occurs from the request task to the view task in order to execute the DATA step view and generate the observations. The DATA step view fills the view buffer with as many observations as will fit.

2. When the view buffer is full, task switching occurs from the view task back to the request task in order to return the requested data. The observations are cleared from the view buffer.

The size of the view buffer determines how many generated observations can be held. The number of generated observations then determines how many times the computer must switch between the request task and the view task. For example, OBSBUF=1 results in task switching for each observation, while OBSBUF=10 results in 10 observations being read into the view buffer at a time. The larger the view buffer is, the less task switching is needed to process a DATA step view, which can speed up execution time.

To improve efficiency, first determine how many observations will fit into the default buffer size, then set the view buffer so that it can hold more generated observations.

Note: Using OBSBUF= can improve processing efficiency by reducing task switching. However, the larger the view buffer size, the more time it takes to fill. This delays the task switching from the view task back to the request task in order to return the requested data. The delay is more apparent in interactive applications. For example, when you use the Viewtable window, the larger the view buffer, the longer it takes to display the requested observations, because the view buffer must be filled before even one observation is returned to the Viewtable. Therefore, before you set a very large view buffer size, consider the type of application that you are using to process the DATA step view as well as the amount of memory that you have available.

Example

For this example, the observation length is 10K, which means that the default view buffer size, which is 32K, would result in three observations at a time to be read into the view buffer. The default view buffer size causes the execution time to be slower, because the computer must do task switching for every three observations that are generated.
To improve performance, the OBSBUF= data set option is set to 100, which causes one hundred observations at a time to be read into the view buffer and reduces task switching in order to process the DATA step view with the PRINT procedure:

```sas
data testview / view=testview;
   ... more SAS statements ...
run;

proc print data=testview (obsbuf=100);
run;
```

See Also

Data Set Options:
  “SPILL= Data Set Option” on page 52

---

**OUTREP= Data Set Option**

Specifies the data representation for the output SAS data set

Valid in: DATA step and PROC steps

Category: Data Set Control

---

**Syntax**

OUTREP=*

**Syntax Description**

format specifies the data representation for the output SAS data set. Data representation is the format in which data is represented in a computer architecture or in an operating environment. For example, on an IBM PC, character data is represented by its ASCII encoding and byte-swapped integers. Native data representation refers to an environment for which the data representation is comparable to the CPU that is accessing the file. For example, a file that is in Windows data representation is native to the Windows operating environment.

Specifying this option enables you to create a SAS data set within the native environment by using a foreign environment data representation. For example, in a UNIX environment, you can create a SAS data set in Windows data representation.

Values for OUTREP= are listed in the following table:
Table 2.2  Data Representation Values for OUTREP= Option

<table>
<thead>
<tr>
<th>OUTREP= Value</th>
<th>Alias*</th>
<th>Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALPHA_TRU64</td>
<td>ALPHA_OSF</td>
<td>Compaq Tru64 UNIX</td>
</tr>
<tr>
<td>ALPHA_VMS_32</td>
<td>ALPHA_VMS</td>
<td>OpenVMS Alpha on 32-bit platform</td>
</tr>
<tr>
<td>ALPHA_VMS_64</td>
<td></td>
<td>OpenVMS Alpha on 64-bit platform</td>
</tr>
<tr>
<td>HP_IA64</td>
<td>HP_ITANIUM</td>
<td>HP-UX on Itanium 64-bit platform</td>
</tr>
<tr>
<td>HP_UX_32</td>
<td>HP_UX</td>
<td>HP-UX on 32-bit platform</td>
</tr>
<tr>
<td>HP_UX_64</td>
<td></td>
<td>HP-UX on 64-bit platform</td>
</tr>
<tr>
<td>INTEL_ABI</td>
<td></td>
<td>ABI UNIX on Intel 32-bit platform</td>
</tr>
<tr>
<td>LINUX_32</td>
<td>LINUX</td>
<td>Linux for Intel Architecture on 32-bit platform</td>
</tr>
<tr>
<td>MIPS_ABI</td>
<td></td>
<td>ABI UNIX on 32-bit platform</td>
</tr>
<tr>
<td>MVS_32</td>
<td>MVS</td>
<td>z/OS on 32-bit platform</td>
</tr>
<tr>
<td>OS2</td>
<td></td>
<td>OS/2 on Intel 32-bit platform</td>
</tr>
<tr>
<td>RS_6000_AIX_32</td>
<td>RS_6000_AIX</td>
<td>AIX UNIX on 32-bit RS/6000</td>
</tr>
<tr>
<td>RS_6000_AIX_64</td>
<td></td>
<td>AIX UNIX on 64-bit RS/6000</td>
</tr>
<tr>
<td>SOLARIS_32</td>
<td>SOLARIS</td>
<td>Sun Solaris on 32-bit platform</td>
</tr>
<tr>
<td>SOLARIS_64</td>
<td></td>
<td>Sun Solaris on 64-bit platform</td>
</tr>
<tr>
<td>VAX_VMS</td>
<td></td>
<td>VAX VMS</td>
</tr>
<tr>
<td>WINDOWS_32</td>
<td>WINDOWS</td>
<td>Microsoft Windows on 32-bit platform</td>
</tr>
<tr>
<td>WINDOWS_64</td>
<td></td>
<td>Microsoft Windows 64-bit Edition</td>
</tr>
</tbody>
</table>

* It is recommended that you use the current values. The aliases are available for compatibility only.

Details

By default, SAS creates a new SAS data set by using the native data representation of the CPU that is running SAS. For example, when using a PC, SAS creates a SAS data set that has ASCII characters and byte-swapped integers.

You can specify the OUTREP= data set option to create a new data set in a foreign data representation. This option enables you to create a SAS data set within the native environment by using a foreign data representation. For example, in a UNIX environment, you can create a SAS data set in Windows data representation.

See Also

Statements:

OUTREP= option in “LIBNAME Statement” on page 1286

POINTOBS= Data Set Option

Controls whether a compressed data set can be processed with random access (by observation number) rather than with sequential access only

Valid in: DATA step and PROC steps
Category: Observation Control
Restriction: POINTOBS= is effective only when creating a compressed data set; otherwise it is ignored.

Syntax

POINTOBS= YES | NO

Syntax Description

YES
causes SAS software to produce a compressed data set that may be randomly accessed by observation number. This is the default.

Examples of accessing data directly by observation number are:

□ the POINT= option of the MODIFY and SET statements in the DATA step
□ going directly to a specific observation number with PROC FSEDIT.

Tip: Specifying POINTOBS=YES does not affect the efficiency of retrieving information from a data set, but it does increase CPU usage by roughly 10% when creating a compressed data set and when updating or adding information to it.

NO
suppresses the ability to randomly access observations in a compressed data set by observation number.

Tip: Specifying POINTOBS=NO is desirable for applications where the ability to point directly to an observation by number within a compressed data set is not important.

If you do not need to access data by observation number, then you can improve performance by roughly 10% when creating a compressed data set and when updating or adding observations to it by specifying POINTOBS=NO.

Details

Note that REUSE=YES takes precedence over POINTOBS=YES. For example:

data test(compress=yes pointobs=yes reuse=yes);

results in a data set that has POINTOBS=NO. Because POINTOBS=YES is the default when you use compression, REUSE=YES causes POINTOBS= to change to NO.

See Also

Data Set Options:
**PW= Data Set Option**

Assigns a read, write, or alter password to a SAS file and enables access to a password-protected SAS file.

**Valid in:** DATA step and PROC steps  
**Category:** Data Set Control

**Syntax**

PW=password

**Syntax Description**

password  
must be a valid SAS name. See “Rules for Words and Names in the SAS Language” in *SAS Language Reference: Concepts*.

**Details**

The PW= option applies to all types of SAS files except catalogs. You can use this option to assign a password to a SAS file or to access a password-protected SAS file.

When replacing a SAS data set that is alter protected, the new data set inherits the alter password. To change the alter password for the new data set, use the MODIFY statement in the DATASETS procedure.

*Note:* A SAS password does not control access to a SAS file beyond the SAS system. You should use the operating system-supplied utilities and file-system security controls in order to control access to SAS files outside of SAS.

**See Also**

Data Set Options:  
“ALTER= Data Set Option” on page 9 
“ENCRYPT= Data Set Option” on page 18 
“READ= Data Set Option” on page 44 
“WRITE= Data Set Option” on page 65
“Manipulating Passwords” in “The DATASETS Procedure” in *Base SAS Procedures Guide*
“File Protection” in *SAS Language Reference: Concepts*

---

**PWREQ= Data Set Option**

Controls the pop up of a requestor window for a data set password

Valid in: DATA and PROC steps
Category: Data Set Control

**Syntax**

PWREQ=YES|NO

**Syntax Description**

YES
specifies that a requestor window appear.

NO
prevents a requestor window from appearing. If a missing or invalid password is entered, the data set is not opened and an error message is written to the SAS log.

**Details**

In an interactive SAS session, the PWREQ= option controls whether a requestor window appears after a user enters an incorrect or a missing password for a SAS data set that is password protected. PWREQ= applies to data sets with read, write, or alter passwords. PWREQ= is most useful in SCL applications.

**See Also**

Data Set Options:
“ALTER= Data Set Option” on page 9
“ENCRYPT= Data Set Option” on page 18
“PW= Data Set Option” on page 43
“READ= Data Set Option” on page 44
“WRITE= Data Set Option” on page 65

---

**READ= Data Set Option**

Assigns a read password to a SAS file and enables access to a read-protected SAS file
RENAME= Data Set Option

Changes the name of a variable

Valid in:    DATA step and PROC steps
Category:   Variable Control

Syntax

RENAME=(old-name-1=new-name-1 < ...old-name-n=new-name-n>)
Syntax Description

old-name
the variable you want to rename.

new-name
the new name of the variable. It must be a valid SAS name.

Details
If you use the RENAME= data set option when you create a data set, the new variable name is included in the output data set. If you use RENAME= on an input data set, the new name is used in DATA step programming statements.

If you use RENAME= on an input data set that is used in a SAS procedure, SAS changes the name of the variable in that procedure. The list of variables to rename must be enclosed in parentheses:

```
proc print data=test(rename=(score1=score2));
```

If you use RENAME= in the same DATA step with either the DROP= or the KEEP= data set option, the DROP= and the KEEP= data set options are applied before RENAME=. Thus, use old-name in the DROP= and KEEP= data set options. You cannot drop and rename the same variable in the same statement.

Comparisons
- The RENAME= data set option differs from the RENAME statement in the following ways:
  - The RENAME= data set option can be used in PROC steps and the RENAME statement cannot.
  - The RENAME statement applies to all output data sets. If you want to rename different variables in different data sets, you must use the RENAME= data set option.
  - To rename variables before processing begins, you must use a RENAME= data set option on the input data set or data sets.
  - Use the RENAME statement or the RENAME= data set option when program logic requires that you rename variables, for example, if two input data sets have variables with the same name. To rename variables as a file management task, use the DATASETS procedure.

Examples

Example 1: Renaming a Variable at Time of Output
This example uses RENAME= in the DATA statement to show that the variable is renamed at the time it is written to the output data set. The variable keeps its original name, X, during the DATA step processing:

```
data two(rename=(x=keys));
   set one;
   z=x+y;
run;
```

Example 2: Renaming a Variable at Time of Input
This example renames variable X to a variable named KEYS in the SET statement, which is a rename before DATA step processing. KEYS, not X, is the name to use for the variable for DATA step processing.
data three;
    set one(rename=(x=keys));
    z=keys+y;
run;

See Also

Data Set Options:
   “DROP= Data Set Option” on page 17
   “KEEP= Data Set Option” on page 30
Statements:
   “RENAME Statement” on page 1379
   “The DATASETS Procedure” in Base SAS Procedures Guide

REPEMPTY= Data Set Option

Controls replacement of like-named temporary or permanent SAS data sets when the new one is empty
Valid in: DATA step and PROC steps
Category: Data Set Control
Restriction: Use with output data sets only.

Syntax

REPEMPTY=YES | NO

Syntax Description

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 B with the new empty data set B that is created by mistake:
       data mylib.a set 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.

**Comparisons**

- For an individual data set, the REPEMPTY= data set option overrides the REPEMPTY= option in the LIBNAME statement.
- The REPEMPTY= and REPLACE= data set options apply to both permanent and temporary SAS data sets. The REPLACE system option, however, only applies to permanent SAS data sets.

**See Also**

Data Set Options:
- “REPLACE= Data Set Option” on page 48
Statement Options:
- REPEMPTY= in the LIBNAME statement on page 1289
System Options:
- “REPLACE System Option” on page 1590

---

**REPLACE= Data Set Option**

Controls replacement of like-named temporary or permanent SAS data sets

- **Valid in:** DATA step and PROC steps
- **Category:** Data Set Control
- **Restriction:** Use with output data sets only.
- **Restriction:** This option is valid only when creating a SAS data set.

**Syntax**

REPLACE=NO | YES

**Syntax Description**

- **NO**
  specifies that a new data set with a given name does not replace an existing data set with the same name.

- **YES**
  specifies that a new data set with a given name replaces an existing data set with the same name.

**Comparisons**

- The REPLACE= data set option overrides the REPLACE system option for the individual data set.
The REPLACE system option only applies to permanent SAS data sets.

**Example**

Using the REPLACE= data set option in this DATA statement prevents SAS from replacing a permanent SAS data set named ONE in a library referenced by MYLIB:

```sas
data mylib.one(replace=no);
```

SAS writes a message in the log that tells you that the file has not been replaced.

**See Also**

System Options:

“REPLACE System Option” on page 1590

---

**REUSE= Data Set Option**

Specifies whether new observations are written to free space in compressed SAS data sets

**Valid in:** DATA step and PROC steps  
**Category:** Data Set Control  
**Restriction:** Use with output data sets only.

**Syntax**

REUSE=NO | YES

**Syntax Description**

**NO**

do not track and reuse space in compressed data sets. New observations are appended to the existing data set. Specifying the NO argument results in less efficient data storage if you delete or update many observations in the SAS data set.

**YES**

tracks and reuses space in compressed SAS data sets. New observations are inserted in the space that is freed when other observations are updated or deleted.

If you plan to use procedures that add observations to the end of SAS data sets (for example, the APPEND and FSEDIT procedures) with compressed data sets, use the REUSE=NO argument. REUSE=YES causes new observations to be added wherever there is space in the file, not necessarily at the end of the file.

**Details**

By default, new observations are appended to existing compressed data sets. If you want to track and reuse free space by deleting or updating other observations, use the REUSE= data set option when you create a compressed SAS data set.
REUSE= has meaning only when you are creating new data sets with the COMPRESS=YES data set option or system option. Using the REUSE= data set option when you are accessing an existing SAS data set has no effect.

**Comparisons**

The REUSE= data set option overrides the REUSE= system option. REUSE=YES takes precedence over POINTOBS=YES. For example, the following statement results in a data set that has POINTOBS=NO:

```sas
data test(compress=yes pointobs=yes reuse=yes);
```

Because POINTOBS=YES is the default when you use compression, REUSE=YES causes POINTOBS= to change to NO.

**See Also**

Data Set Options:

“COMPRESS= Data Set Option” on page 14

System Options:

“REUSE= System Option” on page 1591

---

### SORTEDBY= Data Set Option

**Specifies how the data set is currently sorted**

**Valid in:** DATA step and PROC steps

**Category:** Data Set Control

**Syntax**

SORTEDBY=by-clause < / collate-name> | _NULL_

**Syntax Description**

- **by-clause < / collate-name>**
  
  indicates how the data are currently sorted.
  
  - **by-clause**
    
    names the variables and options that you use in a BY statement in a PROC SORT step.
  
  - **collate-name**
    
    names the collating sequence that is used for the sort. By default, the collating sequence is that of your operating environment. A slash (/) must precede the collating sequence.

  **Operating Environment Information:** For details on collating sequences, see the SAS documentation for your operating environment.

- **_NULL_**
removes any existing sort information.

Details

SAS uses the sort information in these ways:

- For BY-group processing, if the data are already sorted by the BY variable, SAS does not use the index, even if the data set is indexed on the BY variable.
- If an index is selected for WHERE expression processing, the sort information for that data set is changed to reflect the order that is specified by the index.
- At the time you create an index, the sort information can make sorting of key variables unnecessary.
- PROC SQL uses the sort information to process queries more efficiently and to determine whether an internal sort is necessary before performing a join.
- PROC SORT checks for the sort information before it sorts a data set so that data are not resorted unnecessarily.
- PROC SORT sets the sort information whenever it does a sort.

If you update a SAS file in a way that affects the validity of the sort, the sort information is removed. That is, if you change or add any values of the variables by which the data set is sorted, the sort information is removed.

Comparisons

- Use the CONTENTS statement in the DATASETS procedure to see how a data set is sorted.
- The SORTEDBY= option does not cause a data set to be sorted.

Examples

This example uses the SORTEDBY= data set option to specify how the data are currently sorted. The data set ORDERS is sorted by PRIORITY and by the descending values of INDATE. Once the data set is created, the sort information is stored with it. These statements create the data set ORDERS and record the sort information:

```sas
libname mylib 'SAS-data-library';
options yearcutoff=1920;
data mylib.orders(sortedby=priority
    descending indate);
  input priority 1. +1 indate date7.
    +1 office $ code $;
  format indate date7.;
datalines;
1 03may01 CH J8U
1 21mar01 LA M91
1 01dec00 FW L6R
1 27feb99 FW Q2A
2 15jan00 FW I9U
2 09ju199 CH P3Q
3 08apr99 CH H5T
3 31jan99 FW D2W
;```

See Also

The CONTENTS statement in “The DATASETS Procedure” in Base SAS Procedures Guide
“The SORT Procedure” in Base SAS Procedures Guide
“The SQL Procedure” in Base SAS Procedures Guide

SORTSEQ= Data Set Option

Specifies a language-specific collation sequence for the SORT procedure to use for the specified SAS data set

Valid in: DATA step and PROC steps
Category: Data Set Control

SPILL= Data Set Option

Specifies whether to create a spill file for non-sequential processing of a DATA step view

Valid in: DATA step and PROC steps
Category: Data Set Control
Restriction: Valid only for a DATA step view
Default: YES

Syntax

SPILL= YES|NO

Syntax Description

YES
creates a spill file for non-sequential processing of a DATA step view. This is the default.

Interaction: A spill file is never created for sequential processing of a DATA step view.

Tip: A DATA step view that generates large amounts of observations can result in a very large spill file. You must have enough disk space to accommodate the spill file.

NO
does not create a spill file or reduces the size of a spill file.
Interaction: For direct (random) access, a spill file is always created even if you specify SPILL=NO.

Tip: If you do not have enough disk space to accommodate a resulting spill file from a DATA step view that generates a large amount of data, then specify SPILL=NO.

Tip: For SAS procedures that process BY-group data, consider specifying SPILL=NO in order to write only the current BY group to the spill file.

Details

When a DATA step view is opened for non-sequential processing, a spill file is created by default. The spill file contains the observations that are generated by a DATA step view. Subsequent requests for data will read observations from the spill file rather than execute the DATA step view again. The spill file is a temporary file in the WORK library.

Non-sequential processing includes the following access methods, which are supported by several SAS statements and procedures. How the SPILL= data set option operates with each of the access methods is described below:

- **random access** retrieves observations directly either by an observation number or by the value of one or more variables through an index without reading all observations sequentially. Whether SPILL=YES or SPILL=NO, a spill file is always created, because the processing time to restart a DATA step view for each observation would be costly.

- **BY-group access** uses a BY statement to process observations that are ordered, grouped, or indexed according to the values of one or more variables. SPILL=YES creates a spill file the size of all the data that is requested from the DATA step view. SPILL=NO writes only the current BY group to the spill file. The largest size of the spill file will be the size to store the largest BY group.

- **two-pass access** performs multiple sequential passes through the data. With SPILL=NO, no spill file is created. Instead, after the first pass through the data, the DATA step view is restarted for each subsequent pass through the data. If small amounts of data are returned by the DATA step view for each restart, then the processing time to restart the view might become significant.

  Note: With SPILL=NO, subsequent passes through the data could result in generating different data. Some processing might require using a spill file; for example, results from using random functions and computing values that are based on the current time of day could affect the data.

Examples

**Example 1: Using a Spill File for a Small Number of Large BY Groups**  This example creates a DATA step view that generates a large amount of random data, then uses the UNIVARIATE procedure with a BY statement. The example illustrates the effects of SPILL= with a small number of large BY groups.

With SPILL=YES, all observations that are requested from the DATA step view are written to the spill file. With SPILL=NO, only the observations that are in the current BY group are written to the spill file. The information messages that are produced by this example show that the size of the spill file is reduced with SPILL=NO. However, the time to truncate the spill file for each BY group might add to the overall processing time for the DATA step view.
options msglevel=i;

data vw_few_large / view=vw_few_large;
  drop i;
  do byval = 'Group A', 'Group B', 'Group C';
    do i = 1 to 500000;
      r = ranuni(4);
      output;
    end;
  end;
run;

proc univariate data=vw_few_large (spill=yes) noprint;
  var r;
  by byval;
run;

proc univariate data=vw_few_large (spill=no) noprint;
  var r;
  by byval;
run;
### Output 2.8  SAS Log Output

```sas
1  options msglevel=i;
2  data vw_few_large / view=vw_few_large;
3     drop i;
4
5     do byval = 'Group A', 'Group B', 'Group C';
6         do i = 1 to 500000;
7             r = ranuni(4);
8             output;
9         end;
10    end;
11 run;

NOTE: DATA STEP view saved on file WORK.VW_FEW_LARGE.
NOTE: A stored DATA STEP view cannot run under a different operating system.
NOTE: DATA statement used (Total process time):
       real time 21.57 seconds
       cpu time 1.31 seconds

12  proc univariate data=vw_few_large (spill=yes) noprint;
INFO: View WORK.VW_FEW_LARGE open mode: BY-group rewind.
13     var r;
14     by byval;
15  run;

INFO: View WORK.VW_FEW_LARGE opening spill file for output observations.
INFO: View WORK.VW_FEW_LARGE deleting spill file. File size was 22506120 bytes.
NOTE: View WORK.VW_FEW_LARGE.VIEW used (Total process time):
       real time 40.68 seconds
       cpu time 12.71 seconds

16  proc univariate data=vw_few_large (spill=no) noprint;
INFO: View WORK.VW_FEW_LARGE open mode: BY-group rewind.
17     var r;
18     by byval;
19  run;

INFO: View WORK.VW_FEW_LARGE opening spill file for output observations.
INFO: View WORK.VW_FEW_LARGE truncating spill file. File size was 7502040 bytes.
NOTE: The above message was for the following by-group:
       byval=Group A
INFO: View WORK.VW_FEW_LARGE truncating spill file. File size was 7534800 bytes.
NOTE: The above message was for the following by-group:
       byval=Group B
INFO: View WORK.VW_FEW_LARGE truncating spill file. File size was 7534800 bytes.
NOTE: The above message was for the following by-group:
       byval=Group C
INFO: View WORK.VW_FEW_LARGE deleting spill file. File size was 32760 bytes.
NOTE: View WORK.VW_FEW_LARGE.VIEW used (Total process time):
       real time 11.03 seconds
       cpu time 10.95 seconds

NOTE: PROCEDURE UNIVARIATE used (Total process time):
       real time 11.04 seconds
       cpu time 10.96 seconds
```
Example 2: Using a Spill File for a Large Number of Small BY Groups  This example creates a DATA step view that generates a large amount of random data, then uses the UNIVARIATE procedure with a BY statement. This example illustrates the effects of SPILL= with a large number of small BY groups.

With SPILL=YES, all observations that are requested from the DATA step view are written to the spill file. With SPILL=NO, only the observations that are in the current BY group are written to the spill file. The information messages that are produced by this example show that the size of the spill file is reduced with SPILL=NO, and with small BY groups, this results in a large disk space savings.

```plaintext
options msglevel=i;

data vw_many_small / view=vw_many_small;
  drop i;

  do byval = 1 to 100000;
    do i = 1 to 5;
      r = ranuni(4);
      output;
    end;
  end;
run;

proc univariate data=vw_many_small (spill=yes) noprint;
  var r;
  by byval;
run;

proc univariate data=vw_many_small (spill=no) noprint;
  var r;
  by byval;
run;
```
1 options msglevel=i;
2 data vw_many_small / view=vw_many_small;
3   drop i;
4
5   do byval = 1 to 100000;
6     do i = 1 to 5;
7       r = ranuni(4);
8       output;
9     end;
10   end;
11 run;

NOTE: DATA STEP view saved on file WORK.VW_MANY_SMALL.
NOTE: A stored DATA STEP view cannot run under a different operating system.
NOTE: DATA statement used (Total process time):
   real time 0.56 seconds
   cpu time 0.03 seconds

12 proc univariate data=vw_many_small (spill=yes) noprint;
13   INFO: View WORK.VW_MANY_SMALL open mode: BY-group rewind.
14    var r;
15    by byval;
16    run;

INFO: View WORK.VW_MANY_SMALL opening spill file for output observations.
INFO: View WORK.VW_MANY_SMALL deleting spill file. File size was 8024240 bytes.
NOTE: PROCEDURE UNIVARIATE used (Total process time):
   real time 30.73 seconds
   cpu time 29.59 seconds

NOTE: PROCEDURE UNIVARIATE used (Total process time):
   real time 30.96 seconds
   cpu time 29.68 seconds

17 proc univariate data=vw_many_small (spill=no) noprint;
18   INFO: View WORK.VW_MANY_SMALL open mode: BY-group rewind.
19    var r;
20    by byval;
21    run;

INFO: View WORK.VW_MANY_SMALL opening spill file for output observations.
INFO: View WORK.VW_MANY_SMALL truncating spill file. File size was 65504 bytes.
NOTE: The above message was for the following by-group:
   byval=410
INFO: View WORK.VW_MANY_SMALL truncating spill file. File size was 65504 bytes.
NOTE: The above message was for the following by-group:
   byval=819
INFO: View WORK.VW_MANY_SMALL truncating spill file. File size was 65504 bytes.
NOTE: The above message was for the following by-group:
   byval=1229
   . Deleted many INFO and NOTE messages for BY groups
   .
INFO: View WORK.VW_MANY_SMALL truncating spill file. File size was 65504 bytes.
NOTE: The above message was for the following by-group:
   byval=99894
INFO: View WORK.VW_MANY_SMALL deleting spill file. File size was 32752 bytes.
NOTE: PROCEDURE UNIVARIATE used (Total process time):
   real time 29.43 seconds
   cpu time 28.81 seconds

NOTE: PROCEDURE UNIVARIATE used (Total process time):
   real time 29.43 seconds
   cpu time 28.81 seconds
Example 3: Using a Spill File with Two-Pass Access  This examples creates a DATA step view that generates a large amount of random data, then uses the TRANSPOSE procedure. The example illustrates the effects of SPILL= with a procedure that requires two-pass access processing.

When PROC TRANSPOSE processes a DATA step view, the procedure must make two passes through the observations that the view generates. The first pass counts the number of observations, then the second pass performs the transposition. With SPILL=YES, a spill file is created during the first pass, and the second pass reads the observations from the spill file. With SPILL=NO, a spill file is not created—after the first pass, the DATA step view is restarted.

Note that for the first TRANSPOSE procedure, which does not include the SPILL= data set option, even though a spill file is used by default, the informative message about the open mode is not displayed. This occurs to reduce the amount of messages in the SAS log for users who are not using the SPILL= data set option.

```sas
options msglevel=i;
data vw_transpose/view=vw_transpose;
  drop i j;
  array x[10000];
  do i = 1 to 10;
    do j = 1 to dim(x);
      x[j] = ranuni(4);
    end;
  output;
  end;
run;
proc transpose data=vw_transpose out=transposed;
run;
proc transpose data=vw_transpose(spill=yes) out=transposed;
run;
proc transpose data=vw_transpose(spill=no) out=transposed;
run;
```
SAS Data Set Options

### Output 2.10  SAS Log Output

```sas
options msglevel=i;
data vw_transpose/view=vw_transpose;
drop i j;
array x[10000];
do i = 1 to 10;
do j = 1 to dim(x);
x[j] = ranuni(4);
end;
output;
end;
run;

NOTE: DATA STEP view saved on file WORK.VW_TRANSPOSE.
NOTE: A stored DATA STEP view cannot run under a different operating system.
NOTE: DATA statement used (Total process time):
  real time 0.68 seconds
  cpu time 0.18 seconds

proc transpose data=vw_transpose out=transposed;
run;

INFO: View WORK.VW_TRANSPOSE opening spill file for output observations.
INFO: View WORK.VW_TRANSPOSE deleting spill file. File size was 880000 bytes.
NOTE: View WORK.VW_TRANSPOSE used (Total process time):
  real time 2.37 seconds
  cpu time 1.17 seconds

NOTE: There were 10 observations read from the data set WORK.VW_TRANSPOSE.
NOTE: The data set WORK.TRANSPOSE has 10000 observations and 11 variables.
NOTE: PROCEDURE TRANSPOSE used (Total process time):
  real time 4.17 seconds
  cpu time 1.51 seconds

proc transpose data=vw_transpose (spill=yes) out=transposed;
run;

INFO: View WORK.VW_TRANSPOSE open mode: sequential.
INFO: View WORK.VW_TRANSPOSE reopen mode: two-pass.
INFO: View WORK.VW_TRANSPOSE opening spill file for output observations.
INFO: View WORK.VW_TRANSPOSE deleting spill file. File size was 880000 bytes.
NOTE: View WORK.VW_TRANSPOSE used (Total process time):
  real time 0.95 seconds
  cpu time 0.92 seconds

NOTE: There were 10 observations read from the data set WORK.VW_TRANSPOSE.
NOTE: The data set WORK.TRANSPOSE has 10000 observations and 11 variables.
NOTE: PROCEDURE TRANSPOSE used (Total process time):
  real time 1.01 seconds
  cpu time 0.98 seconds

proc transpose data=vw_transpose (spill=no) out=transposed;
run;

INFO: View WORK.VW_TRANSPOSE open mode: sequential.
INFO: View WORK.VW_TRANSPOSE reopen mode: two-pass.
INFO: View WORK.VW_TRANSPOSE restarting for another pass through the data.
NOTE: View WORK.VW_TRANSPOSE used (Total process time):
  real time 1.34 seconds
  cpu time 1.32 seconds

NOTE: The View WORK.VW_TRANSPOSE was restarted 1 times. The following view statistics only apply to the last view restart.
NOTE: There were 10 observations read from the data set WORK.VW_TRANSPOSE.
NOTE: The data set WORK.TRANSPOSE has 10000 observations and 11 variables.
NOTE: PROCEDURE TRANSPOSE used (Total process time):
  real time 1.42 seconds
  cpu time 1.40 seconds
```
TOBSNO= Data Set Option

Specifies the number of observations to be transmitted in each multi-observation exchange with a SAS server

Valid in: DATA step and PROC steps
Category: Data Set Control
Restriction: The TOBSNO= option is valid only for data sets that are accessed through a SAS server via the REMOTE engine.

Syntax
TOBSNO=n

Syntax Description

n
specifies the number of observations to be transmitted.

Details
If the TOBSNO= option is not specified, its value is calculated based on the observation length and the size of the server’s transmission buffers, as specified by the PROC SERVER statement TBUFSIZE= option.

The TOBSNO= option is valid only for data sets that are accessed through a SAS server via the REMOTE engine. If this option is specified for a data set opened for update or accessed via another engine, it is ignored.

See Also
“FOPEN Function” in SAS Component Language: Reference.

TYPE= Data Set Option

Specifies the data set type for a specially structured SAS data set

Valid in: DATA step and PROC steps
Category: Data Set Control
Syntax

TYPE=data-set-type

Syntax Description

data-set-type
    specifies the special type of the data set.

Details

Use the TYPE= data set option in a DATA step to create a special SAS data set in the proper format, or to identify the special type of the SAS data set in a procedure statement.

You can use the CONTENTS procedure to determine the type of a data set.

Most SAS data sets do not have a specified type. However, there are several specially structured SAS data sets that are used by some SAS/STAT procedures. These SAS data sets contain special variables and observations, and they are usually created by SAS statistical procedures. Because most of the special SAS data sets are used with SAS/STAT software, they are described in the SAS/STAT User's Guide.

Other values are available in other SAS software products and are described in the appropriate documentation.

Note: If you use a DATA step with a SET statement to modify a special SAS data set, you must specify the TYPE= option in the DATA statement. The data-set-type is not automatically copied to the data set that is created.

See Also

“Special SAS Data Sets” in SAS/STAT User’s Guide

“The CONTENTS Procedure” in Base SAS Procedures Guide

WHERE= Data Set Option

Selects observations that meet the specified condition

Valid in: DATA step and PROC steps

Category: Observation Control

Restriction: Cannot be used with the POINT= option in the SET and MODIFY statements.

Syntax

WHERE=(where-expression-1<logical-operator where-expression-n>)
Syntax Description

_**where-expression**_

is an arithmetic or logical expression that consists of a sequence of operators, operands, and SAS functions. An operand is a variable, a SAS function, or a constant. An operator is a symbol that requests a comparison, logical operation, or arithmetic calculation. The expression must be enclosed in parentheses.

_**logical-operator**_

can be AND, AND NOT, OR, or OR NOT.

Details

- Use the WHERE= data set option with an input data set to select observations that meet the condition specified in the WHERE expression before SAS brings them into the DATA or PROC step for processing. Selecting observations that meet the conditions of the WHERE expression is the first operation SAS performs in each iteration of the DATA step.
  
  You can also select observations that are written to an output data set. In general, selecting observations at the point of input is more efficient than selecting them at the point of output; however, there are some cases when selecting observations at the point of input is not practical or not possible.

- You can apply OBS= and FIRSTOBS= processing to WHERE processing. For more information see “Processing a Segment of Data That is Conditionally Selected” in _SAS Language Reference: Concepts_.

- You cannot use the WHERE= data set option with the POINT= option in the SET and MODIFY statements.

- If you use both the WHERE= data set option and the WHERE statement in the same DATA step, SAS ignores the WHERE statement for data sets with the WHERE= data set option. However, you can use the WHERE= data set option with the WHERE command in SAS/FSP software.

  _Note:_ Using indexed SAS data sets can improve performance significantly when you are using WHERE expressions to access a subset of the observations in a SAS data set. See “Understanding SAS Indexes” in _SAS Language Reference: Concepts_ for a complete discussion of WHERE expression processing with indexed data sets and a list of guidelines to consider before indexing your SAS data sets.

Comparisons

- The WHERE statement applies to all input data sets, whereas the WHERE= data set option selects observations only from the data set for which it is specified.

- Do not confuse the purpose of the WHERE= data set option. The DROP= and KEEP= data set options select variables for processing, while the WHERE= data set option selects observations.

Examples

**Example 1: Selecting Observations from an Input Data Set**  
This example uses the WHERE= data set option to subset the SALES data set as it is read into another data set:

```sas
data whizmo;
  set sales(where=(product='whizmo'));
```
Example 2: Selecting Observations from an Output Data Set  
This example uses the WHERE= data set option to subset the SALES output data set:

```sas
data whizmo(where=(product='whizmo'));
  set sales;
run;
```

See Also

Statements:

- “WHERE Statement” on page 1421
- “WHERE-Expression Processing” in *SAS Language Reference: Concepts*

WHEREUP= Data Set Option

Specifies whether to evaluate added observations and modified observations against a WHERE expression

Valid in: DATA step and PROC steps

Category: Observation Control

Syntax

```
WHEREUP= NO | YES
```

Syntax Description

**NO**

- does not evaluate added observations and modified observations against a WHERE expression.

**YES**

- evaluates added observations and modified observations against a WHERE expression.

Details

Specify WHEREUP=YES when you want any added observations or modified observations to match a specified WHERE expression.
Examples

Example 1: Accepting Updates That Do Not Match the WHERE Expression

This example shows how WHEREUP= permits observations to be updated and added even though the modified observation does not match the WHERE expression:

```sas
data a;
  x=1;
  output;
  x=2;
  output;
run;

data a;
  modify a(where=(x=1) whereup=no);
  x=3;
  replace; /* Update does not match WHERE expression */
  output; /* Add does not match WHERE expression */
run;
```

In this example, SAS updates the observation and adds the new observation to the data set.

Example 2: Rejecting Updates That Do Not Match the WHERE Expression

In this example, WHEREUP= does not permit observations to be updated or added when the update and the add do not match the WHERE expression:

```sas
data a;
  x=1;
  output;
  x=2;
  output;
run;

data a;
  modify a(where=(x=1) whereup=yes);
  x=3;
  replace; /* Update does not match WHERE expression */
  output; /* Add does not match WHERE expression */
run;
```

In this example, SAS does not update the observation nor does it add the new observation to the data set.

See Also

Data Set Option:

“WHERE= Data Set Option” on page 61
WRITE= Data Set Option

Assigns a write password to a SAS file and enables access to a write-protected SAS file

Valid in: DATA step and PROC steps
Category: Data Set Control

Syntax

WRITE=write-password

Syntax Description

write-password

Details

The WRITE= option applies to all types of SAS files except catalogs. You can use this option to assign a write-password to a SAS file or to access a write-protected SAS file.

Note: A SAS password does not control access to a SAS file beyond the SAS system. You should use the operating system-supplied utilities and file-system security controls in order to control access to SAS files outside of SAS.

See Also

Data Set Options:
“ALTER= Data Set Option” on page 9
“ENCRYPT= Data Set Option” on page 18
“PW= Data Set Option” on page 43
“READ= Data Set Option” on page 44
“Manipulating Passwords” in “The DATASETS Procedure” in Base SAS Procedures Guide
CHAPTER 3

Formats

Definition of Formats 71
Syntax 72
Using Formats 72
  Ways to Specify Formats 72
    PUT Statement 73
    PUT Function 73
    %SYSFUNC 73
    FORMAT Statement 73
    ATTRIB Statement 74
  Permanent versus Temporary Association 74
User-Defined Formats 74
Byte Ordering for Integer Binary Data on Big Endian and Little Endian Platforms 75
  Definitions 75
  How Bytes are Ordered Differently 75
  Writing Data Generated on Big Endian or Little Endian Platforms 75
  Integer Binary Notation and Different Programming Languages 76
Data Conversions and Encodings 77
Working with Packed Decimal and Zoned Decimal Data 77
  Definitions 77
  Types of Data 78
    Packed Decimal Data 78
    Zoned Decimal Data 78
    Packed Julian Dates 78
  Platforms Supporting Packed Decimal and Zoned Decimal Data 79
  Languages Supporting Packed Decimal and Zoned Decimal Data 79
  Summary of Packed Decimal and Zoned Decimal Formats and Informats 80
Formats by Category 81
Dictionary 92
  $ASCIIXw. Format 92
  $BIDIXw. Format 93
  $BINARYXw. Format 93
  $CHARXw. Format 94
  $CPTDWXw. Format 95
  $CPTWDXw. Format 96
  $EBCDICXw. Format 96
  $HEXXw. Format 97
  $KANJIXw. Format 98
  $KANJIXw. Format 98
  $LOGVSW. Format 98
  $LOGVSRw. Format 98
  $MSGCASEXw. Format 99
EURDFMYw. Format 137
EURDFWDXw. Format 138
EURDFWKXw. Format 138
EURFRATSw.d Format 138
EURFRBEFw.d Format 138
EURFRCHFw.d Format 139
EURFRCZKw.d Format 139
EURFRDEMw.d Format 139
EURFRDKKw.d Format 139
EURFRESPw.d Format 139
EURFRFIMw.d Format 140
EURFRFRFw.d Format 140
EURFRGBPw.d Format 140
EURFRGRDw.d Format 140
EURFRHUFw.d Format 141
EURFRIFw.d Format 141
EURFRITLw.d Format 141
EURFRLUFw.d Format 141
EURFRNLGw.d Format 141
EURFRNOKw.d Format 142
EURFRPLZw.d Format 142
EURFRPTEw.d Format 142
EURFRROLw.d Format 142
EURFRRURw.d Format 143
EURFRSEKw.d Format 143
EURFRSITw.d Format 143
EURFRTRLw.d Format 143
EURFRYUDw.d Format 143
EUROw.d Format 144
EUROXw.d Format 144
EURTOATSw.d Format 144
EURTOBEFw.d Format 144
EURTOCHFw.d Format 145
EURTOCZKw.d Format 145
EURTODEMw.d Format 145
EURTODKKw.d Format 145
EURTOESPw.d Format 145
EURTOFIMw.d Format 146
EURTOFRFw.d Format 146
EURTOGBPw.d Format 146
EURTOGRDw.d Format 146
EURTOHUFw.d Format 147
EURTOIEFw.d Format 147
EURTOITLw.d Format 147
EURTOLUFw.d Format 147
EURTONLGu.d Format 147
EURTONOKw.d Format 148
EURTOPLZw.d Format 148
EURTOPTEw.d Format 148
EURTOROLw.d Format 148
EURTOROLw.d Format 148
EURTORURw.d Format 149
EURTOSEKw.d Format 149
EURTOSITw.d Format 149
EURTOTRLw.d Format 149
<table>
<thead>
<tr>
<th>Format</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>EURTOYUDw.d Format</td>
<td>149</td>
</tr>
<tr>
<td>FLOATw.d Format</td>
<td>150</td>
</tr>
<tr>
<td>FRACTw. Format</td>
<td>151</td>
</tr>
<tr>
<td>HDATEw. Format</td>
<td>152</td>
</tr>
<tr>
<td>HEBDATEw. Format</td>
<td>152</td>
</tr>
<tr>
<td>HEXw. Format</td>
<td>152</td>
</tr>
<tr>
<td>HHMMw.d Format</td>
<td>153</td>
</tr>
<tr>
<td>HOURw.d Format</td>
<td>155</td>
</tr>
<tr>
<td>IBw.d Format</td>
<td>156</td>
</tr>
<tr>
<td>IBRw.d Format</td>
<td>158</td>
</tr>
<tr>
<td>IEEEw.d Format</td>
<td>159</td>
</tr>
<tr>
<td>JULDAYw. Format</td>
<td>160</td>
</tr>
<tr>
<td>JULIANw. Format</td>
<td>161</td>
</tr>
<tr>
<td>MINGUOw. Format</td>
<td>162</td>
</tr>
<tr>
<td>MMDDYYw. Format</td>
<td>162</td>
</tr>
<tr>
<td>MMDDYYxw. Format</td>
<td>164</td>
</tr>
<tr>
<td>MMSSw.d Format</td>
<td>166</td>
</tr>
<tr>
<td>MMYYw. Format</td>
<td>167</td>
</tr>
<tr>
<td>MMYYxw. Format</td>
<td>168</td>
</tr>
<tr>
<td>MONNAMEw. Format</td>
<td>170</td>
</tr>
<tr>
<td>MONTHW. Format</td>
<td>171</td>
</tr>
<tr>
<td>MONYYw. Format</td>
<td>172</td>
</tr>
<tr>
<td>NEGPARENw.d Format</td>
<td>173</td>
</tr>
<tr>
<td>NENGOW. Format</td>
<td>174</td>
</tr>
<tr>
<td>NLDATEpw. Format</td>
<td>175</td>
</tr>
<tr>
<td>NLDATEMNw. Format</td>
<td>175</td>
</tr>
<tr>
<td>NLDATEWw. Format</td>
<td>175</td>
</tr>
<tr>
<td>NLDATEWNw. Format</td>
<td>175</td>
</tr>
<tr>
<td>NLDATMw. Format</td>
<td>175</td>
</tr>
<tr>
<td>NLDATMAPw. Format</td>
<td>176</td>
</tr>
<tr>
<td>NLDATMTMw. Format</td>
<td>176</td>
</tr>
<tr>
<td>NLDATMWW. Format</td>
<td>176</td>
</tr>
<tr>
<td>NLMNYw.d Format</td>
<td>176</td>
</tr>
<tr>
<td>NLMNYw.d Format</td>
<td>176</td>
</tr>
<tr>
<td>NLNUMw.d Format</td>
<td>177</td>
</tr>
<tr>
<td>NLNUMIw.d Format</td>
<td>177</td>
</tr>
<tr>
<td>NLPCTw.d Format</td>
<td>177</td>
</tr>
<tr>
<td>NLPCTw.d Format</td>
<td>177</td>
</tr>
<tr>
<td>NLTIMAPw. Format</td>
<td>177</td>
</tr>
<tr>
<td>NLTIMEw. Format</td>
<td>178</td>
</tr>
<tr>
<td>NUMXw.d Format</td>
<td>178</td>
</tr>
<tr>
<td>OCTALw. Format</td>
<td>179</td>
</tr>
<tr>
<td>PDw.d Format</td>
<td>180</td>
</tr>
<tr>
<td>PDJULGw. Format</td>
<td>181</td>
</tr>
<tr>
<td>PDJULlw. Format</td>
<td>182</td>
</tr>
<tr>
<td>PERCENTw.d Format</td>
<td>184</td>
</tr>
<tr>
<td>PIBw.d Format</td>
<td>185</td>
</tr>
<tr>
<td>PIBRw.d Format</td>
<td>186</td>
</tr>
<tr>
<td>PKw.d Format</td>
<td>188</td>
</tr>
<tr>
<td>PVALUEw.d Format</td>
<td>189</td>
</tr>
<tr>
<td>QTRw. Format</td>
<td>190</td>
</tr>
<tr>
<td>QTRRw. Format</td>
<td>191</td>
</tr>
<tr>
<td>RBw.d Format</td>
<td>192</td>
</tr>
<tr>
<td>ROMANw. Format</td>
<td>193</td>
</tr>
</tbody>
</table>
**Definition of Formats**

A format is an instruction that SAS uses to write data values. You use formats to control the written appearance of data values, or, in some cases, to group data values together for analysis. For example, the WORDS22. format, which converts numeric values to their equivalent in words, writes the numeric value 692 as **six hundred ninety-two**.
SAS formats have the following form:

```$><format<w>.<d>```

where

$ indicates a character format; its absence indicates a numeric format.

format

names the format. The format is a SAS format or a user-defined format that was previously defined with the VALUE statement in PROC FORMAT. For more information on user-defined formats, see “The FORMAT Procedure” in Base SAS Procedures Guide.

w

specifies the format width, which for most formats is the number of columns in the output data.

d

specifies an optional decimal scaling factor in the numeric formats.

Formats always contain a period (.) as a part of the name. If you omit the w and the d values from the format, SAS uses default values. The d value that you specify with a format tells SAS to display that many decimal places, regardless of how many decimal places are in the data. Formats never change or truncate the internally stored data values.

For example, in DOLLAR10.2, the w value of 10 specifies a maximum of 10 columns for the value. The d value of 2 specifies that two of these columns are for the decimal part of the value, which leaves eight columns for all the remaining characters in the value. This includes the decimal point, the remaining numeric value, a minus sign if the value is negative, the dollar sign, and commas, if any.

If the format width is too narrow to represent a value, SAS tries to squeeze the value into the space available. Character formats truncate values on the right. Numeric formats sometimes revert to the BESTw.d format. SAS prints asterisks if you do not specify an adequate width. In the following example, the result is x=**.

```x=123;
put x= 2.;```

If you use an incompatible format, such as using a numeric format to write character values, SAS first attempts to use an analogous format of the other type. If this is not feasible, an error message that describes the problem appears in the SAS log.

## Using Formats

### Ways to Specify Formats

You can use formats in the following ways:

- in a PUT statement
- with the PUT, PUTC, or PUTN functions
Formats △ Ways to Specify Formats 73

- with the %SYSFUNC macro function
- in a FORMAT statement in a DATA step or a PROC step
- in an ATTRIB statement in a DATA step or a PROC step.

**PUT Statement**

The PUT statement with a format after the variable name uses a format to write data values in a DATA step. For example, this PUT statement uses the DOLLAR format to write the numeric value for AMOUNT as a dollar amount:

```plaintext
amount=1145.32;
put amount dollar10.2;
```

The DOLLARw.d format in the PUT statement produces this result:

```
$1,145.32
```

See “PUT Statement” on page 1344 for more information.

**PUT Function**

The PUT function writes a numeric variable, a character variable, or a constant with any valid format and returns the resulting character value. For example, the following statement converts the value of a numeric variable into a two-character hexadecimal representation:

```plaintext
num=15;
char=put(num,hex2.);
```

The PUT function creates a character variable named CHAR that has a value of 0F.

The PUT function is useful for converting a numeric value to a character value. See “PUT Function” on page 754 for more information.

%SYSFUNC

The %SYSFUNC (or %QSYSFUNC) macro function executes SAS functions or user-defined functions and applies an optional format to the result of the function outside a DATA step. For example, the following program writes a numeric value in a macro variable as a dollar amount.

```plaintext
%macro tst(amount);
  %put %sysfunc(putn(&amount,dollar10.2));
%mend tst;
%tst (1154.23);
```

For more information, see SAS Macro Language: Reference.

**FORMAT Statement**

The FORMAT statement permanently associates a format with a variable. SAS uses the format to write the values of the variable that you specify. For example, the following statement in a DATA step associates the COMMAw.d numeric format with the variables SALES1 through SALES3:

```plaintext
format sales1-sales3 comma10.2;
```

Because the FORMAT statement permanently associates a format with a variable, any subsequent DATA step or PROC step uses COMMA10.2 to write the values of SALES1, SALES2, and SALES3. See “FORMAT Statement” on page 1209 for more information.
Note: Formats that you specify in a PUT statement behave differently from those that you associate with a variable in a FORMAT statement. The major difference is that formats that are specified in the PUT statement preserve leading blanks. If you assign formats with a FORMAT statement prior to a PUT statement, all leading blanks are trimmed. The result is the same as if you used the colon (:) format modifier. For details about using the colon (:) format modifier, see “PUT Statement, List” on page 1366.

ATTRIB Statement

The ATTRIB statement can also associate a format, as well as other attributes, with one or more variables. For example, in the following statement the ATTRIB statement permanently associates the COMMAw.d format with the variables SALES1 through SALES3:

```plaintext
attrib sales1-sales3 format=comma10.2;
```

Because the ATTRIB statement permanently associates a format with a variable, any subsequent DATA step or PROC step uses COMMA10.2 to write the values of SALES1, SALES2, and SALES3. See “ATTRIB Statement” on page 1111 for more information.

Permanent versus Temporary Association

When you specify a format in a PUT statement, SAS uses the format to write data values during the DATA step but does not permanently associate the format with a variable. To permanently associate a format with a variable, use a FORMAT statement or an ATTRIB statement in a DATA step. SAS permanently associates a format with the variable by modifying the descriptor information in the SAS data set.

Using a FORMAT statement or an ATTRIB statement in a PROC step associates a format with a variable for that PROC step, as well as for any output data sets that the procedure creates that contain formatted variables. For more information on using formats in SAS procedures, see Base SAS Procedures Guide.

User-Defined Formats

In addition to the formats that are supplied with Base SAS software, you can create your own formats. In Base SAS software, PROC FORMAT allows you to create your own formats for both character and numeric variables. For more information, see “The FORMAT Procedure” in Base SAS Procedures Guide.

When you execute a SAS program that uses user-defined formats, these formats should be available. The two ways to make these formats available are

- to create permanent, not temporary, formats with PROC FORMAT
- to store the source code that creates the formats (the PROC FORMAT step) with the SAS program that uses them.

To create permanent SAS formats, see “The FORMAT Procedure” in Base SAS Procedures Guide.

If you execute a program that cannot locate a user-defined format, the result depends on the setting of the FMTERR system option. If the user-defined format is not found, then these system options produce these results:
System Options | Results
--- | ---
FMTERR | SAS produces an error that causes the current DATA or PROC step to stop.
NOFMTERR | SAS continues processing and substitutes a default format, usually the BEST\(w\). or $w\). format.

Although using NOFMTERR enables SAS to process a variable, you lose the information that the user-defined format supplies.
To avoid problems, make sure that your program has access to all user-defined formats that are used.

### Byte Ordering for Integer Binary Data on Big Endian and Little Endian Platforms

#### Definitions

Integer values for binary integer data are typically stored in one of three sizes: one-byte, two-byte, or four-byte. The ordering of the bytes for the integer varies depending on the platform (operating environment) on which the integers were produced.

The ordering of bytes differs between the “big endian” and “little endian” platforms. These colloquial terms are used to describe byte ordering for IBM mainframes (big endian) and for Intel-based platforms (little endian). In the SAS System, the following platforms are considered big endian: AIX, HP-UX, IBM mainframe, Macintosh, and Solaris. The following platforms are considered little endian: OpenVMS Alpha, Digital UNIX, Intel ABI, and Windows.

#### How Bytes are Ordered Differently

On big endian platforms, the value 1 is stored in binary and is represented here in hexadecimal notation. One byte is stored as 01, two bytes as 00 01, and four bytes as 00 00 00 01. On little endian platforms, the value 1 is stored in one byte as 01 (the same as big endian), in two bytes as 01 00, and in four bytes as 01 00 00 00.

If an integer is negative, the “two’s complement” representation is used. The high-order bit of the most significant byte of the integer will be set on. For example, –2 would be represented in one, two, and four bytes on big endian platforms as FE, FF FE, and FF FF FF FE respectively. On little endian platforms, the representation would be FE, FF FF, and FE FF FF FF. These representations result from the output of the integer binary value –2 expressed in hexadecimal representation.

### Writing Data Generated on Big Endian or Little Endian Platforms

SAS can read signed and unsigned integers regardless of whether they were generated on a big endian or a little endian system. Likewise, SAS can write signed
and unsigned integers in both big endian and little endian format. The length of these integers can be up to eight bytes.

The following table shows which format to use for various combinations of platforms. In the Sign? column, “no” indicates that the number is unsigned and cannot be negative. “Yes” indicates that the number can be either negative or positive.

Table 3.1  SAS Formats and Byte Ordering

<table>
<thead>
<tr>
<th>Data created for...</th>
<th>Data written by...</th>
<th>Sign?</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>big endian</td>
<td>big endian</td>
<td>yes</td>
<td>IB or S370FIB</td>
</tr>
<tr>
<td>big endian</td>
<td>big endian</td>
<td>no</td>
<td>PIB, S370FPiB, S370FIBU</td>
</tr>
<tr>
<td>big endian</td>
<td>little endian</td>
<td>yes</td>
<td>S370FIB</td>
</tr>
<tr>
<td>big endian</td>
<td>little endian</td>
<td>no</td>
<td>S370FPiB</td>
</tr>
<tr>
<td>little endian</td>
<td>big endian</td>
<td>yes</td>
<td>IBR</td>
</tr>
<tr>
<td>little endian</td>
<td>little endian</td>
<td>no</td>
<td>PIBR</td>
</tr>
<tr>
<td>little endian</td>
<td>little endian</td>
<td>yes</td>
<td>IB or IBR</td>
</tr>
<tr>
<td>little endian</td>
<td>little endian</td>
<td>no</td>
<td>PIB or PIBR</td>
</tr>
<tr>
<td>big endian</td>
<td>either</td>
<td>yes</td>
<td>S370FIB</td>
</tr>
<tr>
<td>big endian</td>
<td>either</td>
<td>no</td>
<td>S370FPiB</td>
</tr>
<tr>
<td>little endian</td>
<td>either</td>
<td>yes</td>
<td>IBR</td>
</tr>
<tr>
<td>little endian</td>
<td>either</td>
<td>no</td>
<td>PIBR</td>
</tr>
</tbody>
</table>

---

**Integer Binary Notation and Different Programming Languages**

The following table compares integer binary notation according to programming language.

Table 3.2  Integer Binary Notation and Programming Languages

<table>
<thead>
<tr>
<th>Language</th>
<th>2 Bytes</th>
<th>4 Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL/I</td>
<td>FIXED BIN(15)</td>
<td>FIXED BIN(31)</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>INTEGER*2</td>
<td>INTEGER*4</td>
</tr>
<tr>
<td>COBOL</td>
<td>COMP PIC 9(4)</td>
<td>COMP PIC 9(8)</td>
</tr>
</tbody>
</table>
Data Conversions and Encodings

An encoding maps each character in a character set to a unique numeric representation, resulting in a table of all code points. A single character can have different numeric representations in different encodings. For example, the ASCII encoding for the dollar symbol $ is 24hex. The Danish EBCDIC encoding for the dollar symbol $ is 67hex. In order for a version of SAS that normally uses ASCII to properly interpret a data set that is encoded in Danish EBCDIC, the data must be transcoded.

Transcoding is the process of moving data from one encoding to another. When transcoding the ASCII dollar sign to the Danish EBCDIC dollar sign, the hex representation for the character is converted from the value 24 to a 67.

If you want to know the encoding of a particular SAS data set, for SAS 9 and above follow these steps:
1. Locate the data set with SAS Explorer.
2. Right-click the data set.
3. Select Properties from the menu.
4. Click the Details tab.
5. The encoding of the data set is listed, along with other information.

Some situations where data might commonly be transcoded are:
- when you share data between two different SAS sessions that are running in different locales or in different operating environments,
- when you perform text-string operations, such as converting to uppercase or lowercase,
- when you display or print characters from another language,
- when you copy and paste data between SAS sessions running in different locales.

For more information on SAS features designed to handle data conversions from different encodings or operating environments, see SAS National Language Support (NLS): User’s Guide.

Working with Packed Decimal and Zoned Decimal Data

Definitions

Packed decimal specifies a method of encoding decimal numbers by using each byte to represent two decimal digits. Packed decimal representation stores decimal data with exact precision. The fractional part of the number is determined by the informat or format because there is no separate mantissa and exponent.
An advantage of using packed decimal data is that exact precision can be maintained. However, computations involving decimal data might become inexact due to the lack of native instructions.

Zoned decimal specifies a method of encoding decimal numbers in which each digit requires one byte of storage. The last byte contains the number’s sign as well as the last digit. Zoned decimal data produces a printable representation.

Nibble specifies 1/2 of a byte.

Types of Data

Packed Decimal Data

A packed decimal representation stores decimal digits in each “nibble” of a byte. Each byte has two nibbles, and each nibble is indicated by a hexadecimal digit. For example, the value 15 is stored in two nibbles, using the hexadecimal digits 1 and 5.

The sign indication is dependent on your operating environment. On IBM mainframes, the sign is indicated by the last nibble. With formats, C indicates a positive value, and D indicates a negative value. With informats, A, C, E, and F indicate positive values, and B and D indicate negative values. Any other nibble is invalid for signed packed decimal data. In all other operating environments, the sign is indicated in its own byte. If the high-order bit is 1, then the number is negative. Otherwise, it is positive.

The following applies to packed decimal data representation:

- You can use the S370FPD format on all platforms to obtain the IBM mainframe configuration.
- You can have unsigned packed data with no sign indicator. The packed decimal format and informat handles the representation. It is consistent between ASCII and EBCDIC platforms.
- Note that the S370FPDU format and informat expects to have an F in the last nibble, while packed decimal expects no sign nibble.

Zoned Decimal Data

The following applies to zoned decimal data representation:

- A zoned decimal representation stores a decimal digit in the low order nibble of each byte. For all but the byte containing the sign, the high-order nibble is the numeric zone nibble (F on EBCDIC and 3 on ASCII).
- The sign can be merged into a byte with a digit, or it can be separate, depending on the representation. But the standard zoned decimal format and informat expects the sign to be merged into the last byte.
- The EBCDIC and ASCII zoned decimal formats produce the same printable representation of numbers. There are two nibbles per byte, each indicated by a hexadecimal digit. For example, the value 15 is stored in two bytes. The first byte contains the hexadecimal value F1 and the second byte contains the hexadecimal value C5.

Packed Julian Dates

The following applies to packed Julian dates:
The two formats and informats that handle Julian dates in packed decimal representation are PDJULI and PDJULG. PDJULI uses the IBM mainframe year computation, while PDJULG uses the Gregorian computation.

The IBM mainframe computation considers 1900 to be the base year, and the year values in the data indicate the offset from 1900. For example, 98 means 1998, 100 means 2000, and 102 means 2002. 1998 would mean 3898.

The Gregorian computation allows for 2-digit or 4-digit years. If you use 2-digit years, SAS uses the setting of the YEARCUTOFF= system option to determine the true year.

Platforms Supporting Packed Decimal and Zoned Decimal Data

Some platforms have native instructions to support packed and zoned decimal data, while others must use software to emulate the computations. For example, the IBM mainframe has an Add Pack instruction to add packed decimal data, but the Intel-based platforms have no such instruction and must convert the decimal data into some other format.

Languages Supporting Packed Decimal and Zoned Decimal Data

Several different languages support packed decimal and zoned decimal data. The following table shows how COBOL picture clauses correspond to SAS formats and informats.

<table>
<thead>
<tr>
<th>IBM VS COBOL II clauses</th>
<th>Corresponding S370Fxxx formats/informats</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC S9(X) PACKED-DECIMAL</td>
<td>S370FPDw.</td>
</tr>
<tr>
<td>PIC 9(X) PACKED-DECIMAL</td>
<td>S370FPDUw.</td>
</tr>
<tr>
<td>PIC S9(W) DISPLAY</td>
<td>S370FZDw.</td>
</tr>
<tr>
<td>PIC 9(W) DISPLAY</td>
<td>S370FZDUw.</td>
</tr>
<tr>
<td>PIC S9(W) DISPLAY SIGN LEADING</td>
<td>S370FZDLw.</td>
</tr>
<tr>
<td>PIC S9(W) DISPLAY SIGN LEADING SEPARATE</td>
<td>S370FZDSw.</td>
</tr>
<tr>
<td>PIC S9(W) DISPLAY SIGN TRAILING SEPARATE</td>
<td>S370FZDTw.</td>
</tr>
</tbody>
</table>

For the packed decimal representation listed above, X indicates the number of digits represented, and W is the number of bytes. For PIC S9(X) PACKED-DECIMAL, W is \(\text{ceil}\left(\frac{x+1}{2}\right)\). For PIC 9(X) PACKED-DECIMAL, W is \(\text{ceil}\left(\frac{x}{2}\right)\). For example, PIC S9(5) PACKED-DECIMAL represents five digits. If a sign is included, six nibbles are needed. \(\text{ceil}\left(\frac{5+1}{2}\right)\) has a length of three bytes, and the value of W is 3.

Note that you can substitute COMP-3 for PACKED-DECIMAL.

In IBM assembly language, the P directive indicates packed decimal, and the Z directive indicates zoned decimal. The following shows an excerpt from an assembly language listing, showing the offset, the value, and the DC statement:

\[
\text{offset value (in hex) inst label directive}
\]
In PL/I, the FIXED DECIMAL attribute is used in conjunction with packed decimal data. You must use the PICTURE specification to represent zoned decimal data. There is no standardized representation of decimal data for the FORTRAN or the C languages.

### Summary of Packed Decimal and Zoned Decimal Formats and Informats

SAS uses a group of formats and informats to handle packed and zoned decimal data. The following table lists the type of data representation for these formats and informats. Note that the formats and informats that begin with S370 refer to IBM mainframe representation.

<table>
<thead>
<tr>
<th>Format</th>
<th>Type of data representation</th>
<th>Corresponding informat</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>PD</td>
<td>Packed decimal</td>
<td>PD</td>
<td>Local signed packed decimal</td>
</tr>
<tr>
<td>PK</td>
<td>Packed decimal</td>
<td>PK</td>
<td>Unsigned packed decimal; not specific to your operating environment</td>
</tr>
<tr>
<td>ZD</td>
<td>Zoned decimal</td>
<td>ZD</td>
<td>Local zoned decimal</td>
</tr>
<tr>
<td>none</td>
<td>Zoned decimal</td>
<td>ZDB</td>
<td>Translates EBCDIC blank (hex 40) to EBCDIC zero (hex F0), then corresponds to the informat as zoned decimal</td>
</tr>
<tr>
<td>none</td>
<td>Zoned decimal</td>
<td>ZDV</td>
<td>Non-IBM zoned decimal representation</td>
</tr>
<tr>
<td>S370FPD</td>
<td>Packed decimal</td>
<td>S370FPD</td>
<td>Last nibble C (positive) or D (negative)</td>
</tr>
<tr>
<td>S370FPDU</td>
<td>Packed decimal</td>
<td>S370FPDU</td>
<td>Last nibble always F (positive)</td>
</tr>
<tr>
<td>S370FZD</td>
<td>Zoned decimal</td>
<td>S370FZD</td>
<td>Last byte contains sign in upper nibble: C (positive) or D (negative)</td>
</tr>
<tr>
<td>S370FZDU</td>
<td>Zoned decimal</td>
<td>S370FZDU</td>
<td>Unsigned; sign nibble always F</td>
</tr>
<tr>
<td>S370FZDL</td>
<td>Zoned decimal</td>
<td>S370FZDL</td>
<td>Sign nibble in first byte in informat; separate leading sign byte of hex C0 (positive) or D0 (negative) in format</td>
</tr>
<tr>
<td>S370FZDS</td>
<td>Zoned decimal</td>
<td>S370FZDS</td>
<td>Leading sign of - (hex 60) or + (hex 4E)</td>
</tr>
<tr>
<td>S370FZDT</td>
<td>Zoned decimal</td>
<td>S370FZDT</td>
<td>Trailing sign of - (hex 60) or + (hex 4E)</td>
</tr>
<tr>
<td>Format</td>
<td>Type of data representation</td>
<td>Corresponding informat</td>
<td>Comments</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------</td>
<td>------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>PDJULI</td>
<td>Packed decimal</td>
<td>PDJULI</td>
<td>Julian date in packed representation - IBM computation</td>
</tr>
<tr>
<td>PDJULG</td>
<td>Packed decimal</td>
<td>PDJULG</td>
<td>Julian date in packed representation - Gregorian computation</td>
</tr>
<tr>
<td>none</td>
<td>Packed decimal</td>
<td>RMFDUR</td>
<td>Input layout is: mmsstttF</td>
</tr>
<tr>
<td>none</td>
<td>Packed decimal</td>
<td>SHRSTAMP</td>
<td>Input layout is: yyyydddFhmmsssth, where yyyydddF is the packed Julian date; yyyy is a 0-based year from 1900</td>
</tr>
<tr>
<td>none</td>
<td>Packed decimal</td>
<td>SMFSTAMP</td>
<td>Input layout is: xxxxxxxxyyyydddF, where yyyydddF is the packed Julian date; yyyy is a 0-based year from 1900</td>
</tr>
<tr>
<td>none</td>
<td>Packed decimal</td>
<td>PDTIME</td>
<td>Input layout is: 0hhmmssF</td>
</tr>
<tr>
<td>none</td>
<td>Packed decimal</td>
<td>RMFSTAMP</td>
<td>Input layout is: 0hhmmssFyyyydddF, where yyyydddF is the packed Julian date; yyyy is a 0-based year from 1900</td>
</tr>
</tbody>
</table>

---

**Formats by Category**

There are six categories of formats in SAS:

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIDI text handling</td>
<td>instructs SAS to write bidirectional data values from data variables</td>
</tr>
<tr>
<td>Character</td>
<td>instructs SAS to write character data values from character variables.</td>
</tr>
<tr>
<td>Currency Conversion</td>
<td>instructs SAS to convert an amount from one currency to another currency.</td>
</tr>
<tr>
<td>DBCS</td>
<td>instructs SAS to handle various Asian languages.</td>
</tr>
<tr>
<td>Date and Time</td>
<td>instructs SAS to write data values from variables that represent dates,</td>
</tr>
<tr>
<td></td>
<td>times, and datetimes.</td>
</tr>
<tr>
<td>Hebrew text handling</td>
<td>instructs SAS to write Hebrew data from data variables.</td>
</tr>
<tr>
<td>Numeric</td>
<td>instructs SAS to write numeric data values from numeric variables.</td>
</tr>
<tr>
<td>User-Defined</td>
<td>instructs SAS to write data values by using a format that is created with</td>
</tr>
<tr>
<td></td>
<td>PROC FORMAT.</td>
</tr>
</tbody>
</table>
Storing user-defined formats is an important consideration if you associate these formats with variables in permanent SAS data sets, especially those shared with other users. For information on creating and storing user-defined formats, see “The FORMAT Procedure” in *Base SAS Procedures Guide*.

The following table provides brief descriptions of the SAS formats. For more detailed descriptions, see the dictionary entry for each format.

<table>
<thead>
<tr>
<th>Table 3.3 Categories and Descriptions of Formats</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Category</strong></td>
</tr>
<tr>
<td>BIDI text handling</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Character</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Category</td>
</tr>
<tr>
<td>---------------------</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Currency Conversion</td>
</tr>
<tr>
<td>Category</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>“EURFRBEFw.d Format”</td>
</tr>
<tr>
<td>“EURFRCHFw.d Format”</td>
</tr>
<tr>
<td>“EURFRCZKw.d Format”</td>
</tr>
<tr>
<td>“EURFRDEMw.d Format”</td>
</tr>
<tr>
<td>“EURFRDKKw.d Format”</td>
</tr>
<tr>
<td>“EURFRESPw.d Format”</td>
</tr>
<tr>
<td>“EURFRFIMw.d Format”</td>
</tr>
<tr>
<td>“EURFRFRFW.d Format”</td>
</tr>
<tr>
<td>“EURFRESPw.d Format”</td>
</tr>
<tr>
<td>“EURFRGDw.d Format”</td>
</tr>
<tr>
<td>“EURFRHUFw.d Format”</td>
</tr>
<tr>
<td>“EURFRITLw.d Format”</td>
</tr>
<tr>
<td>“EURFRITLw.d Format”</td>
</tr>
<tr>
<td>“EURFRSLUFW.d Format”</td>
</tr>
<tr>
<td>“EURFRNLGw.d Format”</td>
</tr>
<tr>
<td>“EURFRNOKw.d Format”</td>
</tr>
<tr>
<td>“EURFRPLZw.d Format”</td>
</tr>
<tr>
<td>“EURFRLPEw.d Format”</td>
</tr>
<tr>
<td>“EURFRROLw.d Format”</td>
</tr>
<tr>
<td>“EURFRRURw.d Format”</td>
</tr>
<tr>
<td>“EURFRSEKw.d Format”</td>
</tr>
<tr>
<td>“EURFRSITw.d Format”</td>
</tr>
<tr>
<td>Category</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Category</td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Date and Time</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Category</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>“EURDFDEw. Format” on page 136</td>
</tr>
<tr>
<td>“EURDFDNw. Format” on page 137</td>
</tr>
<tr>
<td>“EURDFDITw.d Format” on page 137</td>
</tr>
<tr>
<td>“EURDFDWNw. Format” on page 137</td>
</tr>
<tr>
<td>“EURDFMNW. Format” on page 137</td>
</tr>
<tr>
<td>“EURDFMYw. Format” on page 137</td>
</tr>
<tr>
<td>“EURDFWDXw. Format” on page 138</td>
</tr>
<tr>
<td>“EURDFWXKXw. Format” on page 138</td>
</tr>
<tr>
<td>“HDATEw. Format” on page 152</td>
</tr>
<tr>
<td>“HEBDATExw. Format” on page 152</td>
</tr>
<tr>
<td>“HHMMw.d Format” on page 153</td>
</tr>
<tr>
<td>“HOURw.d Format” on page 155</td>
</tr>
<tr>
<td>“JULDAYw. Format” on page 160</td>
</tr>
<tr>
<td>“JULIANw. Format” on page 161</td>
</tr>
<tr>
<td>“MINGUOW. Format” on page 162</td>
</tr>
<tr>
<td>“MMDDYYw. Format” on page 162</td>
</tr>
<tr>
<td>“MMDDYYYYw. Format” on page 164</td>
</tr>
<tr>
<td>“MMSSw.d Format” on page 166</td>
</tr>
<tr>
<td>“MMYYw. Format” on page 167</td>
</tr>
<tr>
<td>Category</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Category</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>“TODw.d Format”</td>
</tr>
<tr>
<td>“WEEKDATEw. Format”</td>
</tr>
<tr>
<td>“WEEKDATXw. Format”</td>
</tr>
<tr>
<td>“WEEKDAYw. Format”</td>
</tr>
<tr>
<td>“WEEKUw. Format”</td>
</tr>
<tr>
<td>“WEEKVw. Format”</td>
</tr>
<tr>
<td>“WEEKWw. Format”</td>
</tr>
<tr>
<td>“WORDDATEw. Format”</td>
</tr>
<tr>
<td>“WORDDATXw. Format”</td>
</tr>
<tr>
<td>“YEARw. Format”</td>
</tr>
<tr>
<td>“YYMMw. Format”</td>
</tr>
<tr>
<td>“YYMXw. Format”</td>
</tr>
<tr>
<td>“YYMMDDw. Format”</td>
</tr>
<tr>
<td>“YYMMDXw. Format”</td>
</tr>
<tr>
<td>“YYMONw. Format”</td>
</tr>
<tr>
<td>“YYQw. Format”</td>
</tr>
<tr>
<td>“YYQXw. Format”</td>
</tr>
<tr>
<td>“YYQRw. Format”</td>
</tr>
<tr>
<td>Category</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>“YYQRxw. Format” on page 237</td>
</tr>
<tr>
<td>Hebrew text handling</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Numeric</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Category</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Category</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>“S370FPiBw.d Format” on page 200</td>
</tr>
<tr>
<td>“S370FRBw.d Format” on page 202</td>
</tr>
<tr>
<td>“S370FZDw.d Format” on page 203</td>
</tr>
<tr>
<td>“S370FZDLw.d Format” on page 204</td>
</tr>
<tr>
<td>“S370FZDSw.d Format” on page 205</td>
</tr>
<tr>
<td>“S370FZDTw.d Format” on page 206</td>
</tr>
<tr>
<td>“S370FZDUw.d Format” on page 207</td>
</tr>
<tr>
<td>“SSNgw. Format” on page 208</td>
</tr>
<tr>
<td>“VAXRBw.d Format” on page 213</td>
</tr>
<tr>
<td>“w.d Format” on page 214</td>
</tr>
<tr>
<td>“WORDFw. Format” on page 223</td>
</tr>
<tr>
<td>“WORDSw. Format” on page 224</td>
</tr>
<tr>
<td>“YENw.d Format” on page 226</td>
</tr>
<tr>
<td>“Zw.d Format” on page 239</td>
</tr>
<tr>
<td>“ZDw.d Format” on page 240</td>
</tr>
</tbody>
</table>

## Dictionary

### $ASCIIw. Format

**Converts native format character data to ASCII representation**

**Category:** Character  
**Alignment:** left

**Syntax**

$ASCIIw.$
Syntax Description

\( w \)
specifies the width of the output field.

Default: 1
Range: 1–32767

Details
If ASCII is the native format, no conversion occurs.

Comparisons
- On EBCDIC systems, \$ASCIIw. converts EBCDIC character data to ASCIIw.
- On all other systems, \$ASCIIw. behaves like the $CHARw. format.

Examples

```
put x $ascii3.;
```

<table>
<thead>
<tr>
<th>When x = ...</th>
<th>The Result* is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc</td>
<td>616263</td>
</tr>
<tr>
<td>ABC</td>
<td>414243</td>
</tr>
<tr>
<td>()</td>
<td>28293B</td>
</tr>
</tbody>
</table>

* The results are hexadecimal representations of ASCII codes for characters. Each two hexadecimal digits correspond to one byte of binary data, and each byte corresponds to one character.

$BIDIw. Format

Convert a Logical ordered string to a Visually ordered string, and vice versa by reversing the order of Hebrew characters while preserving the order of Latin characters and numbers

Category: Character
Alignment: left

$BINARYw. Format

Converts character data to binary representation
**$CHARw. Format**

Category: Character  
Alignment: left

**Syntax**

$CHARw.

**$CHARw. Format**

Writes standard character data

Category: Character  
Alignment: left

**Syntax**

$CHARw.
Syntax Description

\( w \)

specifies the width of the output field.

**Default:** 8 if the length of variable is undefined; otherwise, the length of the variable

**Range:** 1–32767

Comparisons

- The $\text{CHAR}w.$ format is identical to the $w.$ format.
- The $\text{CHAR}w.$ and $w.$ formats do not trim leading blanks. To trim leading blanks, use the LEFT function to left align character data prior to output, or use the PUT statement with the colon (:) format modifier and the format of your choice to produce list output.
- Use the following table to compare the SAS format $\text{CHAR}8.$ with notation in other programming languages:

<table>
<thead>
<tr>
<th>Language</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAS</td>
<td>$\text{CHAR}8.$</td>
</tr>
<tr>
<td>C</td>
<td>char [8]</td>
</tr>
<tr>
<td>COBOL</td>
<td>PIC x(8)</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>A8</td>
</tr>
<tr>
<td>PL/I</td>
<td>A(8)</td>
</tr>
</tbody>
</table>

Examples

```sas
put @7 name $char4.;
```

When name = ... | The Result is ...
---+----------------|

| XYZ | XYZ |

$\text{CPTDW}w.$ Format

*Writes a character string in Hebrew text that is encoded in IBM-PC (cp862) to Windows Hebrew encoding (cp1255)*

**Category:** Hebrew text handling

**Alignment:** left

**See:** The $\text{CPTDW}$ format in *SAS National Language Support (NLS): User’s Guide*
$\text{CPTWDw. Format}$

Writes a character string that is encoded in Windows (cp1255) to Hebrew DOS (cp862) encoding

Category: Hebrew text handling
Alignment: left
See: The $\text{CPTWD}$ format in *SAS National Language Support (NLS): User’s Guide*

$\text{EBCDICw. Format}$

Converts native format character data to EBCDIC representation

Category: Character
Alignment: left

Syntax

$\text{EBCDICw.}$

Syntax Description

$w$

specifies the width of the output field.

Default: 1
Range: 1–32767

Details

If EBCDIC is the native format, no conversion occurs.

Comparisons

- On ASCII systems, $\text{EBCDICw.}$ converts ASCII character data to EBCDIC.
- On all other systems, $\text{EBCDICw.}$ behaves like the $\text{CHARw.}$ format.

Examples

```plaintext
put name $ebcdic3.;
```
When name = ... The Result* is ...

<table>
<thead>
<tr>
<th>name</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>qrs</td>
<td>9899A2</td>
</tr>
<tr>
<td>QRS</td>
<td>D8D9E2</td>
</tr>
<tr>
<td>+;&gt;</td>
<td>4E5E6E</td>
</tr>
</tbody>
</table>

*The results are shown as hexadecimal representations of EBCDIC codes for characters. Each two hexadecimal digits correspond to one byte of binary data, and each byte corresponds to one character.

---

**$HEXw. Format**

Converts character data to hexadecimal representation

**Category:** Character

**Alignment:** left

**See:** $HEXw. Format in the documentation for your operating environment.

---

**Syntax**

$HEXw.

**Syntax Description**

**w**

specifies the width of the output field.

**Default:** The default width is calculated based on the length of the variable to be printed.

**Range:** 1–32767

**Tip:** To ensure that SAS writes the full hexadecimal equivalent of your data, make $w$ twice the length of the variable or field that you want to represent.

**Tip:** If $w$ is greater than twice the length of the variable that you want to represent, $HEXw.$ pads it with blanks.

---

**Details**

The $HEXw.$ format converts each character into two hexadecimal digits. Each blank counts as one character, including trailing blanks.

---

**Comparisons**

The $HEXw.$ format converts real binary numbers to their hexadecimal equivalent.
Examples

```
put $5 name $hex4.;
```

<table>
<thead>
<tr>
<th>When name = ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>AB</td>
<td>C1C2</td>
</tr>
<tr>
<td>AB</td>
<td>4142</td>
</tr>
</tbody>
</table>

**$KANJIw. Format**

Adds shift-code data to DBCS data

**Category:** DBCS

**Alignment:** left

**See:** The $KANJI format in SAS National Language Support (NLS): User’s Guide

**$KANJIXw. Format**

Removes shift code data from DBCS data

**Category:** DBCS

**Alignment:** left

**See:** The $KANJIX format in SAS National Language Support (NLS): User’s Guide

**$LOGVSw. Format**

Writes a character string in left-to-right logical order to visual order

**Category:** BIDI text handling

**Alignment:** left

**See:** The $LOGVS format in SAS National Language Support (NLS): User’s Guide

**$LOGVSRw. Format**

Writes a character string in right-to-left logical order to visual order
$MSGCASEw. Format

Writes character data in uppercase when the MSGCASE system option is in effect

Category: Character
Alignment: left

Syntax

$MSGCASEw.

Syntax Description

\( w \)

specifies the width of the output field.

Default: 8 if the length of the variable is undefined; otherwise, the length of the variable

Range: 1–32767

Details

When the MSGCASE= system option is in effect, all notes, warnings, and error messages that SAS generates appear in uppercase. Otherwise, all notes, warnings, and error messages appear in mixed case. You specify the MSGCASE= system option in the configuration file or during the SAS invocation.

Operating Environment Information: For more information about the MSGCASE= system option, see the SAS documentation for your operating environment.

Examples

```sas
put name $msgcase.;
```

When name = ...
The Result is ...

| sas | SAS |
$\text{OCTAL}_w. \ \text{Format}$

Converts character data to octal representation

Category: Character
Alignment: left

Syntax

$\text{OCTAL}_w.$

Syntax Description

\(w\)

specifies the width of the output field.

Default: The default width is calculated based on the length of the variable to be printed.

Range: 1–32767

Tip: Because each character value generates three octal characters, increase the value of \(w\) by three times the length of the character value.

Comparisons

The $\text{OCTAL}_w.$ format converts character values to the octal representation of their character codes. The $\text{OCTAL}_w.$ format converts numeric values to octal representation.

Example

The following example shows ASCII output when you use the $\text{OCTAL}_w.$ format.

```
data _null_
  infile datalines truncover;
  input item $5.;
  put item $octal15.;
  datalines;
  art
  rice
  bank
  ;
  run;
```

SAS writes the following results to the log.

```
141162164040040
162151143145040
14214156153040
```
**$QUOTEw. Format**

Writes data values that are enclosed in double quotation marks

*Category:* Character  
*Alignment:* left

**Syntax**

$QUOTEw.

**Syntax Description**

*w*

specifies the width of the output field.

*Default:* 8 if the length of the variable is undefined; otherwise, the length of the variable + 2

*Range:* 2–32767

*Tip:* Make *w* wide enough to include the left and right quotation marks.

**Details**

The following list describes the output that SAS produces when you use the $QUOTEw. format. For examples of these items, see “Examples” on page 102.

- If your data value is not enclosed in quotation marks, SAS encloses the output in double quotation marks.
- If your data value is not enclosed in quotation marks, but the value contains a single quotation mark, SAS
  - encloses the data value in double quotation marks
  - does not change the single quotation mark.
- If your data value begins and ends with single quotation marks, and the value contains double quotation marks, SAS
  - encloses the data value in double quotation marks
  - duplicates the double quotation marks that are found in the data value
  - does not change the single quotation marks.
- If your data value begins and ends with single quotation marks, and contains both double quotation marks and single, contiguous quotation marks, SAS
  - encloses the value in double quotation marks
  - does not change the single quotation marks.
- If your data value begins and ends with single quotation marks, and contains both double quotation marks and single, contiguous quotation marks, SAS
  - encloses the value in double quotation marks
  - duplicates the double quotation marks that are found in the data value
  - does not change the single quotation marks.
If the length of the target field is not large enough to contain the string and its quotation marks, SAS returns all blanks.

**Examples**

```
put name $quote20.;
```

<table>
<thead>
<tr>
<th>When name = ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;SAS&quot;</td>
<td>&quot;SAS&quot;</td>
</tr>
<tr>
<td>&quot;SAS’s&quot;</td>
<td>&quot;SAS’s&quot;</td>
</tr>
<tr>
<td>‘ad”verb”’</td>
<td>‘ad”verb”’</td>
</tr>
<tr>
<td>‘ad’’verb’</td>
<td>‘ad’’verb’</td>
</tr>
<tr>
<td>&quot;ad”’’verb”’</td>
<td>&quot;ad”’’verb”’</td>
</tr>
<tr>
<td>deoxyribonucleotide</td>
<td></td>
</tr>
</tbody>
</table>

**$REVERJw. Format**

Writes character data in reverse order and preserves blanks

**Category:** Character

**Alignment:** right

**Syntax**

$REVERJw.

**Syntax Description**

`w`

specifies the width of the output field.

**Default:** 1 if `w` is not specified

**Range:** 1–32767

**Comparisons**

The $REVERJw. format is similar to the $REVERSw. format except that $REVERSw. left aligns the result by trimming all leading blanks.
Examples

```
put @1 name $revers7.;
```

<table>
<thead>
<tr>
<th>When name* = ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABCD### DCBA</td>
<td>ABCD### DCBA</td>
</tr>
<tr>
<td>###ABCD DCBA</td>
<td>ABCD### DCBA</td>
</tr>
</tbody>
</table>

* The character # represents a blank space.

$\text{\texttt{\$REVERSW. Format}}$

Writes character data in reverse order and left aligns

Category: Character

Alignment: left

Syntax

$\text{\texttt{$\textbf{\$REVERSW.}}}$

Syntax Description

\(w\)

specifies the width of the output field.

Default: 1 if \(w\) is not specified

Range: 1–32767

Comparisons

The $\text{\texttt{$\textbf{\$REVERSW.}}}$ format is similar to the $\text{\texttt{$\textbf{\$REVERJW.}}}$ format except that $\text{\texttt{$\textbf{\$REVERJW.}}}$ does not left align the result.

Examples

```
put @1 name $revers7.;
```
When `name* = ...` The Result is ...

---+----1

| ABCD### | DCBA |
| ####ABCD | DCBA |

* The character # represents a blank space.

---

**$UCS2Bw. Format**

Writes a character string in big-endian, 16-bit, universal character set code in 2 octets (UCS2), Unicode encoding

Category: Character

Alignment: Left


---

**$UCS2BEw. Format**

Writes a big-endian, 16-bit, universal character set code in 2 octets (UCS2) character string in the encoding of the current SAS session

Category: Character

Alignment: left


---

**$UCS2Lw. Format**

Writes data in little-endian, 16-bit, universal character set code in 2 octets (UCS2), Unicode encoding

Category: Character

Alignment: Left


---

**$UCS2LEw. Format**

Writes a character string that is encoded in little-endian, 16-bit, universal character set code in 2 octets (UCS2), in the encoding of the current SAS session
$UCS2Xw. Format

Writes a character string in native-endian, 16-bit, universal character set code in 2 octets (UCS2) Unicode encoding

Category: Character
Alignment: Left

$UCS2XEw. Format

Writes a native-endian, 16-bit, universal character set code in 2 octets (UCS2) character string in the encoding of the current SAS session

Category: Character
Alignment: Left

$UCS4Bw. Format

Writes a character string in big-endian, 32-bit, universal character set code in 4 octets (UCS4), Unicode encoding

Category: Character
Alignment: Left

$UCS4BEw. Format

Writes a big-endian, 32-bit, universal character set code in 4 octets (UCS4), character string in the encoding of the current SAS session

Category: Character
$UCS4Lw. Format

Writes a character string in little-endian, 32-bit, universal character set code in 4 octets (UCS4), Unicode encoding

Category: Character
Alignment: left

$UCS4LEw. Format

Writes a little-endian, 32-bit, universal character set code in 4 octets (UCS4) character string in the encoding of the current SAS session

Category: Character
Alignment: left

$UCS4Xw. Format

Writes a character string in native-endian, 32-bit, universal character set code in 4 octets (UCS4), Unicode encoding

Category: Character
Alignment: left

$UCS4XEw. Format

Writes a native-endian, 32-bit, universal character set code in 4 octets (UCS4) character string in the encoding of the current SAS session

Category: Character
Alignment: left

$UESCw. Format

Writes a character string that is encoded in the current SAS session in Unicode escape (UESC) representation

Category: Character
Alignment: left

$UESCEw. Format

Writes a Unicode escape (UESC) representation character string in the encoding of the current SAS session

Category: Character
Alignment: left

$UNCRw. Format

Writes a character string that is encoded in the current SAS session in numeric character representation (NCR)

Category: Character
Alignment: left

$UNCREw. Format

Writes the numeric character representation (NCR) character string in the encoding of the current SAS session

Category: Character
Alignment: left
$\text{UPAREN}_w. \text{ Format}$

Writes a character string that is encoded in the current SAS session in Unicode parenthesis (UPAREN) representation

Category: Character
Alignment: left

$\text{UPARENE}_w. \text{ Format}$

Writes a character string that is in Unicode parenthesis (UPAREN) in a character string that is encoded in the current SAS session

Category: Character
Alignment: left
See: The $\text{UPARENE}$ format in SAS National Language Support (NLS): User’s Guide

$\text{UPCASE}_w. \text{ Format}$

Converts character data to uppercase

Category: Character
Alignment: left

Syntax

\text{$\text{UPCASE}_w.$}

Syntax Description

\(w\)

specifies the width of the output field.

Default: 8 if the length of the variable is undefined; otherwise, the length of the variable

Range: 1–32767
Details
Special characters, such as hyphens and other symbols, are not altered.

Examples

```
put @1 name $upcase9.;
```

<table>
<thead>
<tr>
<th>When name = ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>coxe-ryan</td>
<td>COXE-RYAN</td>
</tr>
</tbody>
</table>

$UTF8Xw. Format

 Writes a character string in universal transformation format (UTF-8) encoding

Category: Character
Alignment: left

$VARYINGw. Format

 Writes character data of varying length

Valid: in DATA step
Category: Character
Alignment: left

Syntax

```
$VARYINGw. length-variable
```

Syntax Description


\( w \)

specifies the maximum width of the output field for any output line or output file record.

Default: 8 if the length of the variable is undefined; otherwise, the length of the variable
Range: 1–32767

**length-variable**

specifies a numeric variable that contains the length of the current value of the character variable. SAS obtains the value of the length-variable by reading it directly from a field that is described in an INPUT statement, reading the value of a variable in an existing SAS data set, or calculating its value.

**Requirement:** You must specify *length-variable* immediately after $VARYINGw. in a SAS statement.

**Restriction:** *Length-variable* cannot be an array reference.

**Tip:** If the value of *length-variable* is 0, negative, or missing, SAS writes nothing to the output field. If the value of *length-variable* is greater than 0 but less than *w*, SAS writes the number of characters that are specified by *length-variable*. If *length-variable* is greater than or equal to *w*, SAS writes *w* columns.

**Details**

Use $VARYINGw. when the length of a character value differs from record to record. After writing a data value with $VARYINGw., the pointer’s position is the first column after the value.

**Examples**

**Example 1: Obtaining a Variable Length Directly**  
An existing data set variable contains the length of a variable. The data values and the results follow the explanation of this SAS statement:

```sas
put @10 name $varying12. varlen;
```

NAME is a character variable of length 12 that contains values that vary from 1 to 12 characters in length. VARLEN is a numeric variable in the same data set that contains the actual length of NAME for the current observation.

<table>
<thead>
<tr>
<th>When name* = ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>New York 8</td>
<td>New York</td>
</tr>
<tr>
<td>Toronto 7</td>
<td>Toronto</td>
</tr>
<tr>
<td>Buenos Aires 12</td>
<td>Buenos Aires</td>
</tr>
<tr>
<td>Tokyo 5</td>
<td>Tokyo</td>
</tr>
</tbody>
</table>

* The value of NAME appears before the value of VARLEN.

**Example 2: Obtaining a Variable Length Indirectly**  
Use the LENGTH function to determine the length of a variable. The data values and the results follow the explanation of these SAS statements:

```sas
varlen=length(name);
put @10 name $varying12. varlen;
```

The assignment statement determines the length of the varying-length variable. The variable VARLEN contains this length and becomes the *length-variable* argument to the $VARYING12. format.
<table>
<thead>
<tr>
<th>Values*</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>New York</td>
<td>New York</td>
</tr>
<tr>
<td>Toronto</td>
<td>Toronto</td>
</tr>
<tr>
<td>Buenos Aires</td>
<td>Buenos Aires</td>
</tr>
<tr>
<td>Tokyo</td>
<td>Tokyo</td>
</tr>
</tbody>
</table>

* The value of NAME appears before the value of VARLEN.

### $VSLOGw. Format

Writes a character string in visual order to left-to-right logical order

Category: BIDI text handling  
Alignment: left  

### $VSLOGRw. Format

Writes a character string in visual order to right-to-left logical order

Category: BIDI text handling  
Alignment: left  

### $w. Format

Writes standard character data

Category: Character  
Alignment: left  
Alias: $Fw.

Syntax

$w.
Syntax Description

\( w \)

specifies the width of the output field. You can specify a number or a column range.

**Default:** 1 if the length of the variable is undefined; otherwise, the length of the variable

**Range:** 1–32767

Comparisons

The \( $w \). format and the \$CHAR\( w \). format are identical, and they do not trim leading blanks. To trim leading blanks, use the LEFT function to left align character data prior to output, or use list output with the colon (:) format modifier and the format of your choice.

Examples

```sas
put @10 name $5.;
put name $ 10-15;
```

<table>
<thead>
<tr>
<th>Values*</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Cary</td>
<td>Cary</td>
</tr>
<tr>
<td>Tokyo</td>
<td>Tokyo</td>
</tr>
</tbody>
</table>

* The character # represents a blank space.

BEST\( w \). Format

**SAS chooses the best notation**

**Category:** Numeric

**Alignment:** right

**See:** BEST\( w \). Format in the documentation for your operating environment.

Syntax

BEST\( w \).
**Syntax Description**

\( w \)

specifies the width of the output field.

**Default:** 12

**Tip:** If you print numbers between 0 and .01 exclusively, then use a field width of at least 7 to avoid excessive rounding. If you print numbers between 0 and -.01 exclusively, then use a field width of at least 8.

**Range:** 1–32

**Details**

The BEST\( w \) format is the default format for writing numeric values. When there is no format specification, SAS chooses the format that provides the most information about the value according to the available field width. BEST\( w \) rounds the value, and if SAS can display at least one significant digit in the decimal portion, within the width specified, BEST\( w \) produces the result in decimal. Otherwise, it produces the result in scientific notation. SAS always stores the complete value regardless of the format that you use to represent it.

**Examples**

The following statements produce these results.

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>--------</td>
</tr>
<tr>
<td>( x=1257000 ) put x best6.;</td>
<td>1.26E6</td>
</tr>
<tr>
<td>( x=1257000 ) put x best3.;</td>
<td>1E6</td>
</tr>
</tbody>
</table>

**BINARY\( w \). Format**

Converts numeric values to binary representation

**Category:** Numeric

**Alignment:** left

**Syntax**

BINARY\( w \).
**Syntax Description**

\[ w \]

specifies the width of the output field.
- **Default:** 8
- **Range:** 1–64

**Comparisons**

BINARY\[ w \]. converts numeric values to binary representation. The $\text{BINARY} w$. format converts character values to binary representation.

**Examples**

```plaintext
put &1 x binary8.;
```

<table>
<thead>
<tr>
<th>When ( x = ... )</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>----+----1</td>
</tr>
<tr>
<td>123.45</td>
<td>01111011</td>
</tr>
<tr>
<td>123</td>
<td>01111011</td>
</tr>
<tr>
<td>−123</td>
<td>10000101</td>
</tr>
</tbody>
</table>
```

**COMMA\[ w,d \] Format**

Writes numeric values with a comma that separates every three digits and a period that separates the decimal fraction

- **Category:** Numeric
- **Alignment:** right

**Syntax**

COMMA\[ w,d \]

**Syntax Description**

\[ w \]

specifies the width of the output field.
- **Default:** 6
Range: 2–32
Tip: Make w wide enough to write the numeric values, the commas, and the optional decimal point.

\( d \)

optionally specifies the number of digits to the right of the decimal point in the numeric value.

Range: 0–31
Requirement: must be less than \( w \)

Details

The COMMA\( w.d \) format writes numeric values with a comma that separates every three digits and a period that separates the decimal fraction.

Comparisons

- The COMMA\( w.d \) format is similar to the COMMAX\( w.d \) format, but the COMMAX\( w.d \) format reverses the roles of the decimal point and the comma. This convention is common in European countries.
- The COMMA\( w.d \) format is similar to the DOLLAR\( w.d \) format except that the COMMA\( w.d \) format does not print a leading dollar sign.

Examples

```plaintext
put @10 sales comma10.2;
```

<table>
<thead>
<tr>
<th>When sales = ...</th>
<th>The result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>23451.23</td>
<td>23,451.23</td>
</tr>
<tr>
<td>123451.234</td>
<td>123,451.23</td>
</tr>
</tbody>
</table>

See Also

Formats:
- “COMMAX\( w.d \) Format” on page 115
- “DOLLAR\( w.d \) Format” on page 126

COMMAX\( w.d \) Format

Writes numeric values with a period that separates every three digits and a comma that separates the decimal fraction

Category: Numeric
Alignment: right
Syntax
COMMAMAX\(w.d\)

Syntax Description

\(w\)
- specifies the width of the output field.
- **Default:** 6
- **Range:** 2–32
- **Tip:** Make \(w\) wide enough to write the numeric values, the commas, and the optional decimal point.

\(d\)
- optionally specifies the number of digits to the right of the decimal point in the numeric value.
- **Range:** 0–31
- **Requirement:** must be less than \(w\)

Details
The COMMAMAX\(w.d\) format writes numeric values with a period that separates every three digits and with a comma that separates the decimal fraction.

Comparisons
The COMMAMAX\(w.d\) format is similar to the COMMAMAX\(w.d\) format, but the COMMAMAX\(w.d\) format reverses the roles of the decimal point and the comma. This convention is common in European countries.

Examples

```plaintext
put @10 sales commax10.2;
```

<table>
<thead>
<tr>
<th>When sales = ...</th>
<th>The result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>23451.23</td>
<td>23.451,23</td>
</tr>
<tr>
<td>123451.234</td>
<td>123.451,23</td>
</tr>
</tbody>
</table>

**Dw.s Format**

Prints variables, possibly with a great range of values, lining up decimal places for values of similar magnitude.
**Category:** Numeric  
**Alignment:** right

---

**Syntax**  
\texttt{Dw.s}

**Syntax Description**

\textit{w}  
optionally specifies the width of the output field.  
\textbf{Default:} 12  
\textbf{Range:} 1–32

\textit{s}  
optionally specifies the significant digits.  
\textbf{Default:} 3  
\textbf{Range:} 0–16  
\textbf{Requirement:} must be less than \textit{w}

---

**Details**

The \texttt{Dw.s} format writes numbers so that the decimal point aligns in groups of values with similar magnitude.

**Comparisons**

- The \texttt{BESTw.} format writes as many significant digits as possible in the output field, but if the numbers vary in magnitude, the decimal points do not line up.
- \texttt{Dw.s} writes numbers with the desired precision and more alignment than \texttt{BESTw}.
- The \texttt{w.d} format aligns decimal points, if possible, but does not necessarily show the same precision for all numbers.

**Examples**

\begin{verbatim}
put @1 x d10.4;
\end{verbatim}

<table>
<thead>
<tr>
<th>When (x = \ldots)</th>
<th>The Result is \ldots</th>
</tr>
</thead>
</table>
| \hline
| 12345               | 12345.0              |
| 1234.5              | 1234.5               |
| 123.45              | 123.45000            |
| 12.345              | 12.34500             |
| 1.2345              | 1.23450              |
| .12345             | 0.12345             |
| \hline
\end{verbatim}
DATEw. Format

Writes date values in the form \textit{ddmmmyy} or \textit{ddmmmyyyy}

\begin{itemize}
  \item Category: Date and Time
  \item Alignment: right
\end{itemize}

Syntax

\texttt{DATEw.}

Syntax Description

\textit{w}

specifies the width of the output field.

\begin{itemize}
  \item Default: 7
  \item Range: 5–9
  \item Tip: Use a width of 9 to print a 4-digit year.
\end{itemize}

Details

The \texttt{DATEw.} format writes SAS date values in the form \textit{ddmmmyy} or \textit{ddmmmyyyy}, where

\begin{itemize}
  \item \textit{dd} is an integer that represents the day of the month.
  \item \textit{mmm} is the first three letters of the month name.
  \item \textit{yy} or \textit{yyyy} is a two-digit or four-digit integer that represents the year.
\end{itemize}

Examples

The example table uses the input value of 15780, which is the SAS date value that corresponds to March 16, 2003.

\begin{tabular}{ll}
  When the SAS Statement is ... & The Result is ... \\
  \hline
  \texttt{put day date5.;} & 16MAR \\
  \texttt{put day date6.;} & 16MAR \\
  \texttt{put day date7.;} & 16MAR03 \\
\end{tabular}
When the SAS Statement is ... | The Result is ...
---|---
put day date8.; | 16MAR03
put day date9.; | 16MAR2003

**See Also**

Function:

“DATE Function” on page 478

Informat:

“DATEw. Informat” on page 979

**DATEAMP\text{M}w.d Format**

Writes datetime values in the form \textit{ddmmmyy:hh:mm:ss.ss} with AM or PM

Category: Date and Time

Alignment: right

**Syntax**

\textsc{DATEAMP\text{M}w.d}

**Syntax Description**

\textbf{w}

specifies the width of the output field.

Default: 19

Range: 7–40

Tip: SAS requires a minimum \textit{w} value of 13 to write AM or PM. For widths between 10 and 12, SAS writes a 24-hour clock time.

\textbf{d}

optionally specifies the number of digits to the right of the decimal point in the seconds value.

Requirement: must be less than \textit{w}

Range: 0–39

Note: If \textit{w}–\textit{d} < 17, SAS truncates the decimal values. Δ

**Details**

The DATEAMP\text{M}w.d format writes SAS datetime values in the form \textit{ddmmmyy:hh:mm:ss.ss}, where
dd
is an integer that represents the day of the month.

mmm
is the first three letters of the month name.

yy
is a two-digit integer that represents the year.

hh
is an integer that represents the hour.

mm
is an integer that represents the minutes.

ss.ss
is the number of seconds to two decimal places.

Comparisons

The DATEAMPMw.d format is similar to the DATETIMEw.d format except that DATEAMPMw.d prints AM or PM at the end of the time.

Examples

The example table uses the input value of 1347455694, which is the SAS datetime value that corresponds to 11:01:34 AM on April 20, 2003.

<table>
<thead>
<tr>
<th>When the SAS Statement is ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>put event dateampm.;</td>
<td>20APR03:11:01:34 AM</td>
</tr>
<tr>
<td>put event dateampm7.;</td>
<td>20APR03</td>
</tr>
<tr>
<td>put event dateampm10.;</td>
<td>20APR:11</td>
</tr>
<tr>
<td>put event dateampm13.;</td>
<td>20APR03:11 AM</td>
</tr>
<tr>
<td>put event dateampm22.2;</td>
<td>20APR03:11:01:34.00 AM</td>
</tr>
</tbody>
</table>

See Also

Format:

“DATETIMEw.d Format” on page 120

DATETIMEw.d Format

Writes datetime values in the form ddmmmyy:hh:mm:ss.ss

Category: Date and Time
Syntax Description

\( w \)

specifies the width of the output field.

Default: 16

Range: 7–40

Tip: SAS requires a minimum \( w \) value of 16 to write a SAS datetime value with the date, hour, and seconds. Add an additional two places to \( w \) and a value to \( d \) to return values with optional decimal fractions of seconds.

\( d \)

optionally specifies the number of digits to the right of the decimal point in the seconds value.

Requirement: must be less than \( w \)

Range: 0–39

Note: If \( w-d<17 \), SAS truncates the decimal values. △

Details

The DATETIME\( w.d \) format writes SAS datetime values in the form \( ddmmyy:hh:mm:ss.ss \), where

\( dd \)

is an integer that represents the day of the month.

\( mmm \)

is the first three letters of the month name.

\( yy \)

is a two-digit integer that represents the year.

\( hh \)

is an integer that represents the hour in 24–hour clock time.

\( mm \)

is an integer that represents the minutes.

\( ss.ss \)

is the number of seconds to two decimal places.

Examples

The example table uses the input value of 1447213759, which is the SAS datetime value that corresponds to 3:49:19 AM on November 10, 2005.
<table>
<thead>
<tr>
<th>When the SAS Statement is ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>put event datetime.;</td>
<td>10NOV05:03:49:19</td>
</tr>
<tr>
<td>put event datetime7.;</td>
<td>10NOV05</td>
</tr>
<tr>
<td>put event datetime12.;</td>
<td>10NOV05:03</td>
</tr>
<tr>
<td>put event datetime18.;</td>
<td>10NOV05:03:49:19</td>
</tr>
<tr>
<td>put event datetime18.1;</td>
<td>10NOV05:03:49:19.0</td>
</tr>
<tr>
<td>put event datetime19.;</td>
<td>10NOV2005:03:49:19</td>
</tr>
<tr>
<td>put event datetime20.1;</td>
<td>10NOV2005:03:49:19.0</td>
</tr>
<tr>
<td>put event datetime21.2;</td>
<td>10NOV2005:03:49:19.00</td>
</tr>
</tbody>
</table>

**See Also**

Formats:
- “DATEw. Format” on page 118
- “TIMEw.d Format” on page 209

Function:
- “DATETIME Function” on page 480

Informats:
- “DATEw. Informat” on page 979
- “DATETIMEw. Informat” on page 980
- “TIMEw. Informat” on page 1037

---

**DAYw. Format**

*Writes date values as the day of the month*

**Category:** Date and Time  
**Alignment:** right

**Syntax**

`DAYw.`

**Syntax Description**

`w`

specifies the width of the output field.  
**Default:** 2
Range: 2–32

Examples

The example table uses the input value of 16601, which is the SAS date value that corresponds to June 14, 2005.

<table>
<thead>
<tr>
<th>When the SAS Statement is ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>put date day2.;</td>
<td>14</td>
</tr>
</tbody>
</table>

DDMMYYw. Format

Writesc date values in the form \(ddm<y>yy\) or \(dll/mm<y>yy\), where a forward slash is the separator and the year appears as either 2 or 4 digits.

Category: Date and Time
Alignment: right

Syntax

DDMMYYw.

Syntax Description

\(w\)

specifies the width of the output field.

Default: 8

Range: 2–10

Interaction: When \(w\) has a value of from 2 to 5, the date appears with as much of the day and the month as possible. When \(w\) is 7, the date appears as a two-digit year without slashes.

Details

The DDMMYYw. format writes SAS date values in the form \(ddm<y>yy\) or \(dll/mm<y>yy\), where

\(dd\)

is an integer that represents the day of the month.

/ is the separator.
\textit{mm}

is an integer that represents the month.

\textit{<yy>yy}

is a two-digit or four-digit integer that represents the year.

**Examples**

The following examples use the input value of 16794, which is the SAS date value that corresponds to December 24, 2005.

<table>
<thead>
<tr>
<th>When the SAS Statement is ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>put date ddmmyy5.;</code></td>
<td>24/12</td>
</tr>
<tr>
<td><code>put date ddmmyy6.;</code></td>
<td>241205</td>
</tr>
<tr>
<td><code>put date ddmmyy7.;</code></td>
<td>241205</td>
</tr>
<tr>
<td><code>put date ddmmyy8.;</code></td>
<td>24/12/05</td>
</tr>
<tr>
<td><code>put date ddmmyy10.;</code></td>
<td>24/12/2005</td>
</tr>
</tbody>
</table>

**See Also**

Formats:

- “DATE\textit{w}. Format” on page 118
- “DDMMYY\textit{xw}. Format” on page 124
- “MMDDYY\textit{w}. Format” on page 162
- “YYMMDD\textit{w}. Format” on page 229

Function:

- “MDY Function” on page 643

Informats:

- “DATE\textit{w}. Informat” on page 979
- “DDMMYY\textit{xw}. Informat” on page 982
- “MMDDYY\textit{w}. Informat” on page 993
- “YYMMDD\textit{w}. Informat” on page 1044

**DDMMYY\textit{xw}. Format**

Writes date values in the form \textit{ddm}<\textit{yy}-\textit{yy} or \textit{ddXmmX}<\textit{yy}-\textit{yy}, where \textit{X} represents a specified separator and the year appears as either 2 or 4 digits

*Category:* Date and Time

*Alignment:* right
Syntax

DDMMYYxw.

Syntax Description

\( x \)

identifies a separator or specifies that no separator appear between the day, the month, and the year. Valid values for \( x \) are:
- B separates with a blank
- C separates with a colon
- D separates with a dash
- N indicates no separator
- P separates with a period
- S separates with a slash.

\( w \)

specifies the width of the output field.
- Default: 8
- Range: 2–10

Interaction: When \( w \) has a value of from 2 to 5, the date appears with as much of the day and the month as possible. When \( w \) is 7, the date appears as a two-digit year without separators.

Interaction: When \( x \) has a value of N, the width range changes to 2–8.

Details

The DDMMYYxw. format writes SAS date values in the form \( ddmm<yy>yy \) or \( ddXmmX<yy>yy \), where

\( dd \)

is an integer that represents the day of the month.

\( X \)

is a specified separator.

\( mm \)

is an integer that represents the month.

\( <yy>yy \)

is a two-digit or four-digit integer that represents the year.

Examples

The following examples use the input value of 16511, which is the SAS date value that corresponds to March 16, 2005.
DOLLAR\(w.d\) Format

Writers numeric values with a leading dollar sign, a comma that separates every three digits, and a period that separates the decimal fraction

**Category:** Numeric

**Alignment:** right

### Syntax

\texttt{DOLLARw.d}

### Syntax Description

\(w\)

 specifies the width of the output field.

**Default:** 6

**Range:** 2–32
\( d \)

optionally specifies the number of digits to the right of the decimal point in the numeric value.

**Range:** 0–31

**Requirement:** must be less than \( w \)

### Details

The DOLLAR\( w.d \) format writes numeric values with a leading dollar sign, a comma that separates every three digits, and a period that separates the decimal fraction.

The hexadecimal representation of the code for the dollar sign character (\$) is 5B on EBCDIC systems and 24 on ASCII systems. The monetary character that these codes represent might be different in other countries, but DOLLAR\( w.d \) always produces one of these codes. If you need another monetary character, define your own format with the FORMAT procedure. See the “The FORMAT Procedure” in *Base SAS Procedures Guide* for more details.

### Comparisons

- The DOLLAR\( w.d \) format is similar to the DOLLARX\( w.d \) format, but the DOLLARX\( w.d \) format reverses the roles of the decimal point and the comma. This convention is common in European countries.
- The DOLLAR\( w.d \) format is the same as the COMMA\( w.d \) format except that the COMMA\( w.d \) format does not write a leading dollar sign.

### Examples

```sas
put @3 netpay dollar10.2;
```

<table>
<thead>
<tr>
<th>When netpay = ...</th>
<th>The result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1254.71</td>
<td>$1,254.71</td>
</tr>
</tbody>
</table>

### See Also

**Formats:**
- “COMMA\( w.d \) Format” on page 114
- “DOLLARX\( w.d \) Format” on page 127

### DOLLARX\( w.d \) Format

Writes numeric values with a leading dollar sign, a period that separates every three digits, and a comma that separates the decimal fraction.

**Category:** Numeric
Alignment: right

Syntax

DOLLARXw.d

Syntax Description

\( w \)
specifies the width of the output field.

Default: 6
Range: 2–32

\( d \)
optionally specifies the number of digits to the right of the decimal point in the numeric value.

Default: 6
Range: 2–31
Requirement: must be less than \( w \)

Details

The DOLLARXw.d format writes numeric values with a leading dollar sign, with a period that separates every three digits, and with a comma that separates the decimal fraction.

The hexadecimal representation of the code for the dollar sign character ($) is 5B on EBCDIC systems and 24 on ASCII systems. The monetary character that these codes represent might be different in other countries, but DOLLARXw.d always produces one of these codes. If you need another monetary character, define your own format with the FORMAT procedure. See “The FORMAT Procedure” in Base SAS Procedures Guide for more details.

Comparisons

- The DOLLARXw.d format is similar to the DOLLARw.d format, but the DOLLARXw.d format reverses the roles of the decimal point and the comma. This convention is common in European countries.
- The DOLLARXw.d format is the same as the COMMAXw.d format except that the COMMAw.d format does not write a leading dollar sign.

Examples

```
put @3 netpay dollarx10.2;
```

<table>
<thead>
<tr>
<th>When netpay = ...</th>
<th>The result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1254.71</td>
<td>$1,254.71</td>
</tr>
</tbody>
</table>

See Also

Formats:

“COMMAXw.d Format” on page 115
“DOLLARw.d Format” on page 126

DOWNAMEw. Format

Writes date values as the name of the day of the week

Category: Date and Time
Alignment: right

(Syntax)

DOWNAMEw.

(Syntax Description)

w
specifies the width of the output field.

Default: 9
Range: 1–32
Tip: If you omit w, SAS prints the entire name of the day.

Details

If necessary, SAS truncates the name of the day to fit the format width. For example, the DOWNAME2. prints the first two letters of the day name.

Examples

The example table uses the input value of 13589, which is the SAS date value that corresponds to March 16, 1997.

<table>
<thead>
<tr>
<th>When the SAS Statement is ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>put date downame.;</td>
<td>Sunday</td>
</tr>
</tbody>
</table>


See Also

Format:
“WEEKDAYw. Format” on page 219

DTDATEw. Format

Expects a datetime value as input and writes date values in the form `ddmmmyy` or `ddmmmyyyy`

Category: Date and Time
Alignment: right

Syntax

**DTDATEw.**

Syntax Description

\[ w \]

- specifies the width of the output field.
- **Default:** 7
- **Range:** 5–9
- **Tip:** Use a width of 9 to print a 4-digit year.

Details

The `DTDATEw.` format writes SAS date values in the form `ddmmmyy` or `ddmmmyyyy`, where

- **dd**
  - is an integer that represents the day of the month.
- **mmm**
  - are the first three letters of the month name.
- **yy** or **yyyy**
  - is a two-digit or four-digit integer that represents the year.

Comparisons

The `DTDATEw.` format produces the same type of output that the `DATEw.` format produces. The difference is that the `DTDATEw.` format requires a datetime value.

Examples

The example table uses a datetime value of 16APR2000:10:00:00 as input, and prints both a two-digit and a four-digit year for the `DTDATEw.` format.
When the SAS statement is ... | The Result is ...
---|---
put trip_date=dtdate.; | 16APR00
put trip_date=dtdate9.; | 16APR2000

**See Also**

Formats:

“DATEw. Format” on page 118

**DTMONYYw. Format**

Writes the date part of a datetime value as the month and year in the form *mmmyy* or *mmmyyyy*

**Category:** Date and Time

**Alignment:** right

**Syntax**

DTMONYYw.

**Syntax Description**

*w*

specifies the width of the output field.

**Default:** 5

**Range:** 5–7

**Details**

The DTMONYYw. format writes SAS datetime values in the form *mmmyy* or *mmmyyyy*, where

*mmm*

is the first three letters of the month name.

*yy* or *yyyy*

is a two-digit or four-digit integer that represents the year.

**Comparisons**

The DTMONYYw. format and the MONYYw. format are similar in that they both write date values. The difference is that DTMONYYw. expects a datetime value as input, and MONYYw. expects a SAS date value.
Examples

The example table uses as input the value 1476598132, which is the SAS datetime value that corresponds to October 16, 2006, at 06:08:52 AM.

| When the SAS Statement is ... | The Result is ...
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>put date dtmonyy.;</td>
<td>OCT06</td>
</tr>
<tr>
<td>put date dtmonyy5.;</td>
<td>OCT06</td>
</tr>
<tr>
<td>put date dtmonyy6.;</td>
<td>OCT06</td>
</tr>
<tr>
<td>put date dtmonyy7.;</td>
<td>OCT2006</td>
</tr>
</tbody>
</table>

See Also

Formats:

“DATETIMEw.d Format” on page 120
“MONYYw. Format” on page 172

DTWKDATXw. Format

Writes the date part of a datetime value as the day of the week and the date in the form day-of-week, dd month-name yy (or yyyy)

Category: Date and Time

Alignment: right

Syntax

DTWKDATXw.

Syntax Description

\( w \)

specifies the width of the output field.

Default: 29

Range: 3–37

Details

The DTWKDATXw. format writes SAS date values in the form day-of-week, dd month-name, yy or yyyy, where
day-of-week
    is either the first three letters of the day name or the entire day name.

dd
    is an integer that represents the day of the month.

month-name
    is either the first three letters of the month name or the entire month name.

yy or yyyy
    is a two-digit or four-digit integer that represents the year.

Comparisons
The DTWKDATXw. format is similar to the WEEKDATXw. format in that they both write date values. The difference is that DTWKDATXw. expects a datetime value as input, and WEEKDATXw. expects a SAS date value.

Examples
The example table uses as input the value 1476598132, which is the SAS datetime value that corresponds to October 16, 2002, at 06:08:52 AM.

<table>
<thead>
<tr>
<th>When the SAS Statement is ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>put date dtwkdatx.;</td>
<td>Monday, 16 October 2006</td>
</tr>
<tr>
<td>put date dtwkdatx3.;</td>
<td>Mon</td>
</tr>
<tr>
<td>put date dtwkdatx8.;</td>
<td>Mon</td>
</tr>
<tr>
<td>put date dtwkdatx25.;</td>
<td>Monday, 16 Oct 2006</td>
</tr>
</tbody>
</table>

See Also

Formats:
“DATETIMEw.d Format” on page 120
“WEEKDATXw. Format” on page 217

DTYEARw. Format

Writes the date part of a datetime value as the year in the form yy or yyyy

Category:  Date and Time
Alignment:  right

Syntax

DTYEARw.
Syntax Description

\(w\)

specifies the width of the output field.

Default: 4
Range: 2–4

Comparisons

The DTYYQC\(w\). format is similar to the YEAR\(w\). format in that they both write date values. The difference is that DTYYQC\(w\). expects a datetime value as input, and YEAR\(w\). expects a SAS date value.

Examples

The example table uses as input the value 1476598132, which is the SAS datetime value that corresponds to October 16, 2006, at 06:08:52 AM.

<table>
<thead>
<tr>
<th>When the SAS Statement is ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>put date dtyear.;</td>
<td>2006</td>
</tr>
<tr>
<td>put date dtyear2.;</td>
<td>06</td>
</tr>
<tr>
<td>put date dtyear3.;</td>
<td>06</td>
</tr>
<tr>
<td>put date year4.;</td>
<td>2006</td>
</tr>
</tbody>
</table>

See Also

Formats:
“DATETIME\(w.d\) Format” on page 120
“YEAR\(w\). Format” on page 225

DTYYQC\(w\). Format

Writes the date part of a datetime value as the year and the quarter and separates them with a colon (:\)

Category: Date and Time
Alignment: right

Syntax

DTYYQC\(w\).
Syntax Description

\[ w \]

specifies the width of the output field.

**Default:** 4

**Range:** 4–6

Details

The \texttt{DTYYQC\textit{w}}. format writes SAS datetime values in the form \textit{yy} or \textit{yyyy}, followed by a colon (:) and the numeric value for the quarter of the year.

Examples

The example table uses as input the value 1476598132, which is the SAS datetime value that corresponds to October 16, 2006, at 06:08:52 PM.

<table>
<thead>
<tr>
<th>When the SAS Statement is ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{put date dtyyqc;}</td>
<td>06:4</td>
</tr>
<tr>
<td>\texttt{put date dtyyqc4;}</td>
<td>06:4</td>
</tr>
<tr>
<td>\texttt{put date dtyyqc5;}</td>
<td>06:4</td>
</tr>
<tr>
<td>\texttt{put date dtyyqc6;}</td>
<td>2006:4</td>
</tr>
</tbody>
</table>

See Also

- Formats:
  - “\texttt{DATETIME}\textit{w.d} Format” on page 120

\textbf{Ew. Format}

\textit{Ew.} Format

\textit{Ew}. Format

\textit{Ew.} Format in the documentation for your operating environment.

Syntax

\textit{Ew}.
Syntax Description

\( w \)

specifies the width of the output field.

**Default:** 12  
**Range:** 7–32

Details

SAS reserves the first column of the result for a minus sign.

Examples

```plaintext
put @1 x e10.;
```

<table>
<thead>
<tr>
<th>When ( x = \ldots )</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1257</td>
<td>1.257E+03</td>
</tr>
<tr>
<td>-1257</td>
<td>-1.257E+03</td>
</tr>
</tbody>
</table>

**EURDFDDw. Format**

Writes international date values in the form `dd.mm.yy` or `dd.mm.yyyy`

**Category:** Date and Time  
**Alignment:** right

**See:** The EURDFDD format in *SAS National Language Support (NLS): User’s Guide*

**EURDFDEw. Format**

Writes international date values in the form `ddmmmyy` or `ddmmmyyyy`

**Category:** Date and Time  
**Alignment:** right

**See:** The EURDFDE format in *SAS National Language Support (NLS): User’s Guide*
**EURDFDNw. Format**

Writes international date values as the day of the week

Category: Date and Time
Alignment: right

**EURDFDTw.d Format**

Writes international datetime values in the form *ddmmmyy:hh:mm:ss.ss* or *ddmmmyyyy hh:mm:ss.ss*

Category: Date and Time
Alignment: right

**EURDFDWNw. Format**

Writes international date values as the name of the day

Category: Date and Time
Alignment: right

**EURDFMNw. Format**

Writes international date values as the name of the month

Category: Date and Time
Alignment: right

**EURDFMYw. Format**

Writes international date values in the form *mmmyy* or *mmmyyyy*
EURDFWDXw. Format

Writes international date values as the name of the month, the day, and the year in the form *dd month-name yy* (or *yyyy*)

Category: Date and Time
Alignment: right

EURDFWKXw. Format

Writes international date values as the name of the day and date in the form *day-of-week, dd month-name yy* (or *yyyy*)

Category: Date and Time
Alignment: right

EURFRATSw.d Format

Converts an amount from Austrian schillings to euros

Category: Currency Conversion
Alignment: right

EURFRBEFw.d Format

Converts an amount from Belgian francs to euros

Category: Currency Conversion
Alignment: right
**EURFRCHFw.d Format**

Converts an amount from Swiss francs to euros  
Category: Currency Conversion  
Alignment: right  

---

**EURFRCZKw.d Format**

Converts an amount from Czech koruny to euros  
Category: Currency Conversion  
Alignment: right  

---

**EURFRDEMw.d Format**

Converts an amount from Deutsche marks to euros  
Category: Currency Conversion  
Alignment: right  

---

**EURFRDKKw.d Format**

Converts an amount from Danish kroner to euros  
Category: Currency Conversion  
Alignment: right  

---

**EURFRESPw.d Format**

Converts an amount from Spanish pesetas to euros
Category: Currency Conversion
Alignment: right

**EURFIMw.d Format**

Converts an amount from Finnish markkaa to euros

Category: Currency Conversion
Alignment: right

**EURFRFw.d Format**

Converts an amount from French francs to euros

Category: Currency Conversion
Alignment: right

**EURFRGBPw.d Format**

Converts an amount from British pounds to euros

Category: Currency Conversion
Alignment: right

**EURFRGRDw.d Format**

Converts an amount from Greek drachmas to euros

Category: Currency Conversion
Alignment: right
EURFRHUFw.d Format

Converts an amount from Hungarian forints to euros
Category: Currency Conversion
Alignment: right

EURFRIEPw.d Format

Converts an amount from Irish pounds to euros
Category: Currency Conversion
Alignment: right

EURFRITLw.d Format

Converts an amount from Italian lire to euros
Category: Currency Conversion
Alignment: right

EURFRLUFw.d Format

Converts an amount from Luxembourg francs to euros
Category: Currency Conversion
Alignment: right

EURFRNLGw.d Format

Converts an amount from Dutch guilders to euros
Category: Currency Conversion
Alignment: right

---

**EURFRNOK**\text{}_{w.d} Format

Converts an amount from Norwegian krone to euros

Category: Currency Conversion
Alignment: right

---

**EURFRPLZ**\text{}_{w.d} Format

Converts an amount from Polish zlotys to euros

Category: Currency Conversion
Alignment: right

---

**EURFRPTE**\text{}_{w.d} Format

Converts an amount from Portuguese escudos to euros

Category: Currency Conversion
Alignment: right

---

**EURFRROL**\text{}_{w.d} Format

Converts an amount from Romanian lei to euros

Category: Currency Conversion
Alignment: right
EURFRURw.d Format

Converts an amount from Russian rubles to euros

Category: Currency Conversion
Alignment: right

EURFRSEKw.d Format

Converts an amount from Swedish kronor to euros

Category: Currency Conversion
Alignment: right

EURFRSITw.d Format

Converts an amount from Slovenian tolars to euros

Category: Currency Conversion
Alignment: right

EURFRTRLw.d Format

Converts an amount from Turkish liras to euros

Category: Currency Conversion
Alignment: right

EURFRYUDw.d Format

Converts an amount from Yugoslavian dinars to euros
EUROw.d Format

Writes numeric values with a leading euro symbol (E), a comma that separates every three digits, and a period that separates the decimal fraction

Category: Numeric
Alignment: right

EUROXw.d Format

Writes numeric values with a leading euro symbol (E), a period that separates every three digits, and a comma that separates the decimal fraction

Category: Numeric
Alignment: right

EURTOATSw.d Format

Converts an amount in euros to Austrian schillings

Category: Currency Conversion
Alignment: right

EURTOBEFw.d Format

Converts an amount in euros to Belgian francs

Category: Currency Conversion
Alignment: right
**EURTOCHF**\textit{w.d} Format

Converts an amount in euros to Swiss francs

Category: Currency Conversion  
Alignment: right  

---

**EURTOCZK**\textit{w.d} Format

Converts an amount in euros to Czech koruny

Category: Currency Conversion  
Alignment: right  

---

**EURTODEM**\textit{w.d} Format

Converts an amount in euros to Deutsche marks

Category: Currency Conversion  
Alignment: right  

---

**EURTODKK**\textit{w.d} Format

Converts an amount in euros to Danish kroner

Category: Currency Conversion  
Alignment: right  

---

**EURTOESP**\textit{w.d} Format

Converts an amount in euros to Spanish pesetas
**EURTOFIMw.d Format**

Converts an amount in euros to Finnish markkaa

Category: Currency Conversion
Alignment: right

**EURTOFRFw.d Format**

Converts an amount in euros to French francs

Category: Currency Conversion
Alignment: right

**EURTOGBPw.d Format**

Converts an amount in euros to British pounds

Category: Currency Conversion
Alignment: right

**EURTOGRDw.d Format**

Converts an amount in euros to Greek drachmas

Category: Currency Conversion
Alignment: right
**EURTOHUF**<i>W.d</i> Format

Converts an amount in euros to Hungarian forints

Category: Currency Conversion
Alignment: right

**EURTOIEP**<i>W.d</i> Format

Converts an amount in euros to Irish pounds

Category: Currency Conversion
Alignment: right

**EURTOITL**<i>W.d</i> Format

Converts an amount in euros to Italian lire

Category: Currency Conversion
Alignment: right

**EURTOLUF**<i>W.d</i> Format

Converts an amount in euros to Luxembourg francs

Category: Currency Conversion
Alignment: right

**EURTONLG**<i>W.d</i> Format

Converts an amount in euros to Dutch guilders
EURTONOKw.d Format

Converts an amount in euros to Norwegian krone

Category: Currency Conversion
Alignment: right

EURTOPLZw.d Format

Converts an amount in euros to Polish zlotys

Category: Currency Conversion
Alignment: right

EURTOPTEw.d Format

Converts an amount in euros to Portuguese escudos

Category: Currency Conversion
Alignment: right

EURTOROLw.d Format

Converts an amount in euros to Romanian lei

Category: Currency Conversion
Alignment: right
EURTORUR\textit{w.d} Format

Converts an amount in euros to Russian rubles

Category: Currency Conversion
Alignment: right

EURTOSEK\textit{w.d} Format

Converts an amount in euros to Swedish kronor

Category: Currency Conversion
Alignment: right

EURTOSIT\textit{w.d} Format

Converts an amount in euros to Slovenian tolars

Category: Currency Conversion
Alignment: right

EURTOTRL\textit{w.d} Format

Converts an amount in euros to Turkish liras

Category: Currency Conversion
Alignment: right

EURTOYUD\textit{w.d} Format

Converts an amount in euros to Yugoslavian dinars
FLOAT\(w.d\) Format

Generates a native single-precision, floating-point value by multiplying a number by \(10^{d}\) raised to the \(d\)th power.

**Syntax**

\[
\text{FLOAT}\(w.d\)
\]

**Syntax Description**

\(w\)

- Specifies the width of the output field.
- **Requirement**: width must be 4

\(d\)

- Optionally specifies the power of 10 by which to divide the value.

**Details**

This format is useful in operating environments where a float value is not the same as a truncated double. Values that are written by FLOAT4 typically are those meant to be read by some other external program that runs in your operating environment and that expects these single-precision values.

**Note:** If the value that is to be formatted is a missing value, or if it is out-of-range for a native single-precision, floating-point value, a single-precision value of zero is generated.

On IBM mainframe systems, a four-byte floating-point number is the same as a truncated eight-byte floating-point number. However, in operating environments using the IEEE floating-point standard, such as IBM PC-based operating environments and most UNIX operating environments, a four-byte floating-point number is not the same as a truncated double. Hence, the RB4 format does not produce the same results as the FLOAT4 format. Floating-point representations other than IEEE may have this same characteristic.

**Comparisons**

The following table compares the names of float notation in several programming languages:
Language | Float Notation
---|---
SAS | FLOAT4
FORTRAN | REAL+4
C | float
IBM 370 ASM | E
PL/I | FLOAT BIN(21)

Examples

```
put x float4.;
```

<table>
<thead>
<tr>
<th>When x = ...</th>
<th>The Result* is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>\texttt{3F800000}</td>
</tr>
</tbody>
</table>

* The result is a hexadecimal representation of a binary number that is stored in IEEE form.

**FRACTw. Format**

Converts numeric values to fractions

- **Category:** Numeric
- **Alignment:** right

**Syntax**

```
FRACTw.
```

**Syntax Description**

\(w\)

- specifies the width of the output field.
- **Default:** 10
- **Range:** 4–32

**Details**

Dividing the number 1 by 3 produces the value 0.33333333. To write this value as 1/3, use the FRACTw. format. FRACTw. writes fractions in reduced form, that is, 1/2 instead of 50/100.
Examples

```plaintext
put x fract8.;
```

<table>
<thead>
<tr>
<th>x</th>
<th>The Result is</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.6666666667</td>
<td>2/3</td>
</tr>
<tr>
<td>0.2784</td>
<td>174/625</td>
</tr>
</tbody>
</table>

**HDATEw. Format**

Writes date values in the form `yyyy mmmmm dd` where `dd` is the day-of-the-month, `mmmmm` represents the month’s name in Hebrew, and `yyyy` is the year

Category: Date and Time
Alignment: right

**HEBDATEw. Format**

Writes date values according to the Jewish calendar

Category: Date and Time
Alignment: right

**HEXw. Format**

Converts real binary (floating-point) values to hexadecimal representation

Category: Numeric
Alignment: left
See: HEXw. Format in the documentation for your operating environment.

**Syntax**

`HEXw.`
Syntax Description

\( w \)

specifies the width of the output field.

**Default:** 8

**Range:** 1–16

**Tip:** If \( w < 16 \), the HEX\( w \) format converts real binary numbers to fixed-point integers before writing them as hexadecimal digits. It also writes negative numbers in two's complement notation, and right aligns digits. If \( w = 16 \), HEX\( w \) displays floating-point values in their hexadecimal form.

Details

In any operating environment, the least significant byte written by HEX\( w \) is the rightmost byte. Some operating environments store integers with the least significant digit as the first byte. The HEX\( w \) format produces consistent results in any operating environment regardless of the order of significance by byte.

*Note:* Different operating environments store floating-point values in different ways. However, the HEX16 format writes hexadecimal representations of floating-point values with consistent results in the same way that your operating environment stores them.

Comparisons

The HEX\( w \) numeric format and the $HEXw$ character format both generate the hexadecimal equivalent of values.

Examples

```
p @8 x hex8;
```

<table>
<thead>
<tr>
<th>When ( x = \ldots )</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>------</strong></td>
<td><strong>--------</strong></td>
</tr>
<tr>
<td>35.4</td>
<td>00000023</td>
</tr>
<tr>
<td>88</td>
<td>00000058</td>
</tr>
<tr>
<td>2.33</td>
<td>00000002</td>
</tr>
<tr>
<td>-150</td>
<td>FFFFFFF6A</td>
</tr>
</tbody>
</table>

HHMM\( w.d \) Format

**Writes time values as hours and minutes in the form hh:mm**

**Category:** Date and Time
Alignment: right

Syntax
HHMM\textsubscript{w}.\textsubscript{d}

Syntax Description

\textit{w}

specifies the width of the output field.

\textbf{Default}: 5

\textbf{Range}: 2–20

\textit{d}

optionally specifies the number of digits to the right of the decimal point in the minutes value.

\textbf{Requirement}: must be less than \textit{w}

\textbf{Range}: 1–19

Details

The HHMM\textsubscript{w}.\textsubscript{d} format writes SAS datetime values in the form \textit{hh:mm}, where

\textit{hh}

is an integer.

\textit{mm}

is the number of minutes that range from 00 through 59.

SAS rounds hours and minutes that are based on the value of seconds in a SAS time value.

Comparisons

The HHMM\textsubscript{w}.\textsubscript{d} format is similar to the TIME\textsubscript{w}.\textsubscript{d} format except that the HHMM\textsubscript{w}.\textsubscript{d} format does not print seconds.

Examples

The example table uses the input value of 46796, which is the SAS time value that corresponds to 12:59:56 PM.

<table>
<thead>
<tr>
<th>When the SAS statement is ...</th>
<th>The result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>put time hhmm.;</td>
<td>13:00</td>
</tr>
</tbody>
</table>

SAS rounds up the time value four seconds based on the value of seconds in the SAS time value.
See Also

Formats:
“HOURw.d Format” on page 155
“MMSSw.d Format” on page 166
“TIMEw.d Format” on page 209

Functions:
“HMS Function” on page 578
“HOUR Function” on page 579
“MINUTE Function” on page 646
“SECOND Function” on page 820
“TIME Function” on page 856

Informat:
“TIMEw. Informat” on page 1037

HOURw.d Format

Writes time values as hours and decimal fractions of hours

Category: Date and Time
Alignment: right

Syntax

HOURw.d

Syntax Description

w
specifies the width of the output field.
Default: 2
Range: 2–20

d
optionally specifies the number of digits to the right of the decimal point in the hour value. Therefore, SAS prints decimal fractions of the hour.
Requirement: must be less than w
Range: 0–19

Details
SAS rounds hours based on the value of minutes in the SAS time value.
Examples

The example table uses the input value of 41400, which is the SAS time value that corresponds to 11:30 AM.

<table>
<thead>
<tr>
<th>When the SAS Statement is ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>put time hour4.1;</td>
<td>11.5</td>
</tr>
</tbody>
</table>

See Also

Formats:
- “HHMMw.d Format” on page 153
- “MMSSw.d Format” on page 166
- “TIMEw.d Format” on page 209
- “TODw.d Format” on page 212

Functions:
- “HMS Function” on page 578
- “HOUR Function” on page 579
- “MINUTE Function” on page 646
- “SECOND Function” on page 820
- “TIME Function” on page 856

Informat:
- “TIMEw. Informat” on page 1037

IBw.d Format

Writes native integer binary (fixed-point) values, including negative values

Category: Numeric
Alignment: left
See: IBw.d Format in the documentation for your operating environment.

Syntax

IBw.d

Syntax Description

w

specifies the width of the output field.
**Default:** 4  
**Range:** 1–8

\(d\)

optionally specifies to multiply the number by \(10^d\).

**Details**

The IB\(w.d\) format writes integer binary (fixed-point) values, including negative values that are represented in two's complement notation. IB\(w.d\) writes integer binary values with consistent results if the values are created in the same type of operating environment that you use to run SAS.

*Note:* Different operating environments store integer binary values in different ways. This concept is called byte ordering. For a detailed discussion about byte ordering, see “Byte Ordering for Integer Binary Data on Big Endian and Little Endian Platforms” on page 75.

**Comparisons**

The IB\(w.d\) and PIB\(w.d\) formats are used to write native format integers. (Native format allows you to read and write values created in the same operating environment.) The IBR\(w.d\) and PIBR\(w.d\) formats are used to write little endian integers in any operating environment.

To view a table that shows the type of format to use with big endian and little endian integers, see Table 3.1 on page 76.

To view a table that compares integer binary notation in several programming languages, see Table 3.2 on page 76.

**Examples**

```r
y=put(x,ib4.);
put y $hex8.;
```

<table>
<thead>
<tr>
<th>x</th>
<th>The Result on Big Endian Platforms*</th>
<th>The Result on Little Endian Platforms* is</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>00000080</td>
<td>80000000</td>
</tr>
</tbody>
</table>

* The result is a hexadecimal representation of a four-byte integer binary number. Each byte occupies one column of the output field.

**See Also**

Format:

“IBR\(w.d\) Format” on page 158
IBRw.d Format

Writes integer binary (fixed-point) values in Intel and DEC formats

**Category:** Numeric
**Alignment:** left

**Syntax**

`IBRw.d`

**Syntax Description**

`w`

specifies the width of the output field.

**Default:** 4
**Range:** 1–8

`d`

optionally specifies to multiply the number by 10^d.

**Details**

The IBRw.d format writes integer binary (fixed-point) values, including negative values that are represented in two’s complement notation. IBRw.d writes integer binary values that are generated by and for Intel and DEC operating environments. Use IBRw.d to write integer binary data from Intel or DEC environments on other operating environments. The IBRw.d format in SAS code allows for a portable implementation for writing the data in any operating environment.

**Note:** Different operating environments store integer binary values in different ways. This concept is called byte ordering. For a detailed discussion about byte ordering, see “Byte Ordering for Integer Binary Data on Big Endian and Little Endian Platforms” on page 75.

**Comparisons**

- The IBw.d and PIBw.d formats are used to write native format integers. (Native format allows you to read and write values that are created in the same operating environment.)
- The IBRw.d and PIBRw.d formats are used to write little endian integers, regardless of the operating environment you are writing on.
- In Intel and DEC operating environments, the IBw.d and IBRw.d formats are equivalent.

To view a table that shows the type of format to use with big endian and little endian integers, see Table 3.1 on page 76.

To view a table that compares integer binary notation in several programming languages, see Table 3.2 on page 76.
Examples

```plaintext
y=put(x,ibr4.);
put y $hex8.;
```

<table>
<thead>
<tr>
<th>When x = ...</th>
<th>The Result* is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>80000000</td>
</tr>
</tbody>
</table>

* The result is a hexadecimal representation of a 4-byte integer binary number. Each byte occupies one column of the output field.

See Also

Format: “IBw.d Format” on page 156

**IEEEw.d Format**

Generates an IEEE floating-point value by multiplying a number by 10 raised to the $d$th power

**Category:** Numeric

**Alignment:** left

**Syntax**

```
IEEEw.d
```

**Syntax Description**

**$w$**

specifies the width of the output field.

**Default:** 8

**Range:** 3–8

**Tip:** If $w$ is 8, an IEEE double-precision, floating-point number is written. If $w$ is 5, 6, or 7, an IEEE double-precision, floating-point number is written, which assumes truncation of the appropriate number of bytes. If $w$ is 4, an IEEE single-precision floating-point number is written. If $w$ is 3, an IEEE single-precision, floating-point number is written, which assumes truncation of one byte.

**$d$**

optionally specifies to multiply the number by $10^d$. 
Details

This format is useful in operating environments where IEEE\textit{w}.d is the floating-point representation that is used. In addition, you can use the IEEE\textit{w}.d format to create files that are used by programs in operating environments that use the IEEE floating-point representation.

Typically, programs generate IEEE values in single-precision (4 bytes) or double-precision (8 bytes). Programs perform truncation solely to save space on output files. Machine instructions require that the floating-point number be one of the two lengths. The IEEE\textit{w}.d format allows other lengths, which enables you to write data to files that contain space-saving truncated data.

Examples

```plaintext
test1=put(x,ieee4.);
put test1 $hex8.;

test2=put(x,ieee5.);
put test2 $hex10.;
```

<table>
<thead>
<tr>
<th>When (x = \ldots)</th>
<th>The Result* is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3F800000</td>
</tr>
<tr>
<td></td>
<td>3FF0000000</td>
</tr>
</tbody>
</table>

* The result contains hexadecimal representations of binary numbers stored in IEEE form.

---

**JULDAY\textit{w}. Format**

* writes date values as the Julian day of the year

**Category:** Date and Time

**Alignment:** right

**Syntax**

\texttt{JULDAY\textit{w}.}

**Syntax Description**

\(\textit{w}\)

specifies the width of the output field.

**Default:** 3

**Range:** 3–32
Details

The JULDAYw. format writes SAS date values in the form ddd, where

$$ddd$$

is the number of the day, 1–365 (or 1–366 for leap years).

Examples

The example table uses the input values of 13515, which is the SAS date value that corresponds to January 1, 1997, and 13589, which is the SAS date value that corresponds to March 16, 1997.

<table>
<thead>
<tr>
<th>When the SAS Statement is ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>put date julday3.;</td>
<td>1</td>
</tr>
<tr>
<td>put date julday3.;</td>
<td>75</td>
</tr>
</tbody>
</table>

JULIANw. Format

Writes date values as Julian dates in the form yyd or yyyydd

Category:  Date and Time
Alignment:  left

Syntax

JULIANw.

Syntax Description

$$w$$

specifies the width of the output field.

Default:  5

Range:  5–7

Tip:  If $$w$$ is 5, the JULIANw. format writes the date with a two-digit year. If $$w$$ is 7, the JULIANw. format writes the date with a four-digit year.

Details

The JULIANw. format writes SAS date values in the form yyd or yyyydd, where

$$yy$$ or $$yyyy$$

is a two-digit or four-digit integer that represents the year.
is the number of the day, 1–365 (or 1–366 for leap years), in that year.

**Examples**

The example table uses the input value of 16794, which is the SAS date value that corresponds to December 24, 2005 (the 358th day of the year).

<table>
<thead>
<tr>
<th>When the SAS Statement is ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>put date julian5.;</td>
<td>05358</td>
</tr>
<tr>
<td>put date julian7.;</td>
<td>2005358</td>
</tr>
</tbody>
</table>

**See Also**

Functions:

“DATEJUL Function” on page 479
“JULDATE Function” on page 613

Informat:

“JULIANw. Informat” on page 992

---

**MINGUOw. Format**

Writes date values as Taiwanese dates in the form **yyymmmdd**

**Category:** Date and Time

**Alignment:** left

**See:** The MINGUO format in *SAS National Language Support (NLS): User’s Guide*

---

**MMDDYYw. Format**

Writes date values in the form **mmdd<yy>yy** or **mm/dd/<yy>yy**, where a forward slash is the separator and the year appears as either 2 or 4 digits

**Category:** Date and Time

**Alignment:** right

**Syntax**

MMDDYYw.
**Syntax Description**

\( w \)

specifies the width of the output field.

**Default:** 8

**Range:** 2–10

**Interaction:** When \( w \) has a value of from 2 to 5, the date appears with as much of the month and the day as possible. When \( w \) is 7, the date appears as a two-digit year without slashes.

**Details**

The MMDDYYw. format writes SAS date values in the form \( mmdd<yy>yy \) or \( mm/dd<yy>yy \), where

- \( mm \) is an integer that represents the month.
- / is the separator.
- \( dd \) is an integer that represents the day of the month.
- \( <yy>yy \) is a two-digit or four-digit integer that represents the year.

**Examples**

The following examples use the input value of 16734, which is the SAS date value that corresponds to October 25, 2005.

<table>
<thead>
<tr>
<th>When the SAS Statement is ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>put day mmddyy2.;</td>
<td>10</td>
</tr>
<tr>
<td>put day mmddyy3.;</td>
<td>10</td>
</tr>
<tr>
<td>put day mmddyy4.;</td>
<td>1025</td>
</tr>
<tr>
<td>put day mmddyy5.;</td>
<td>10/25</td>
</tr>
<tr>
<td>put day mmddyy6.;</td>
<td>102505</td>
</tr>
<tr>
<td>put day mmddyy7.;</td>
<td>102505</td>
</tr>
<tr>
<td>put day mmddyy8.;</td>
<td>10/25/05</td>
</tr>
<tr>
<td>put day mmddyy10.;</td>
<td>10/25/2005</td>
</tr>
</tbody>
</table>
See Also

Formats:
“DATEw. Format” on page 118
“DDMMYYw. Format” on page 123
“MMDDYYxw. Format” on page 164
“YYMMDDw. Format” on page 229

Functions:
“DAY Function” on page 481
“MDY Function” on page 643
“MONTH Function” on page 655
“YEAR Function” on page 916

Informats:
“DATEw. Informat” on page 979
“DDMMYYw. Informat” on page 982
“YYMMDDw. Informat” on page 1044

MMDDYYxw. Format

Writes date values in the form *mmdd<yy>yy* or *mmXddX<yy>yy*, where X represents a specified separator and the year appears as either 2 or 4 digits.

Category: Date and Time
Alignment: right

Syntax

MMDDYYxw.

Syntax Description

x
identifies a separator or specifies that no separator appear between the month, the day, and the year. Valid values for x are:

B
separates with a blank

C
separates with a colon

D
separates with a dash

N
indicates no separator
P
separates with a period
S
separates with a slash.

\(w\)
specifies the width of the output field.

Default: 8

Range: 2–10

Interaction: When \(w\) has a value of from 2 to 5, the date appears with as much of the month and the day as possible. When \(w\) is 7, the date appears as a two-digit year without separators.

Interaction: When \(x\) has a value of N, the width range changes to 2–8.

Details

The **MMDDYYxw.** format writes SAS date values in the form \(mmdd<yy>yy\) or \(mmXddX<yy>yy\), where

- \(mm\) is an integer that represents the month.
- \(X\) is a specified separator.
- \(dd\) is an integer that represents the day of the month.
- \(<yy>yy\) is a two-digit or four-digit integer that represents the year.

Examples

The following examples use the input value of 16731, which is the SAS date value that corresponds to October 22, 2005.

<table>
<thead>
<tr>
<th>When the SAS Statement is ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>put day mmddyy5.;</td>
<td>10:22</td>
</tr>
<tr>
<td>put day mmddyyd8.;</td>
<td>10–22–05</td>
</tr>
<tr>
<td>put day mmddyp10.;</td>
<td>10.22.2005</td>
</tr>
<tr>
<td>put day mmddyyn8.;</td>
<td>10222005</td>
</tr>
</tbody>
</table>
See Also

Formats:
- “DATEw. Format” on page 118
- “DDMMYYxw. Format” on page 124
- “MMDDYYw. Format” on page 162
- “YYMMDDxw. Format” on page 230

Functions:
- “DAY Function” on page 481
- “MDY Function” on page 643
- “MONTH Function” on page 655
- “YEAR Function” on page 916

Informat:
- “MMDDYYw. Informat” on page 993

---

**MMSSw.d Format**

Writes time values as the number of minutes and seconds since midnight

**Category:** Date and Time

**Alignment:** right

---

**Syntax**

**MMSSw.d**

**Syntax Description**

**w**

specifies the width of the output field.

Default: 5

Range: 2–20

Tip: Set w to a minimum of 5 to write a value that represents minutes and seconds.

**d**

optionally specifies the number of digits to the right of the decimal point in the seconds value. Therefore, the SAS time value includes fractional seconds.

Range: 0–19

Restriction: must be less than w

**Examples**

The example table uses the input value of 4530.
When the SAS Statement is ...

<table>
<thead>
<tr>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>put time mmss.;</td>
</tr>
</tbody>
</table>

**See Also**

Formats:
- “HHMMw.d Format” on page 153
- “TIMEw.d Format” on page 209

Functions:
- “HMS Function” on page 578
- “MINUTE Function” on page 646
- “SECOND Function” on page 820

Informat:
- “TIMEw. Informat” on page 1037

**MMYYw. Format**

Writes date values in the form *mmM<yy>yy*, where *M* is the separator and the year appears as either 2 or 4 digits

**Category:** Date and Time  
**Alignment:** right

**Syntax**

**MMYYw.**

**Syntax Description**

\[ w \]

specifies the width of the output field.  

**Default:** 7  
**Range:** 5–32  
**Interaction:** When *w* has a value of 5 or 6, the date appears with only the last two digits of the year. When *w* is 7 or more, the date appears with a four-digit year.

**Details**

The MMYYw. format writes SAS date values in the form *mmM<yy>yy*, where
\[ mm \]
is an integer that represents the month.

\[ M \]
is the character separator.

\[ <yy>yy \]
is a two-digit or four-digit integer that represents the year.

### Examples

The following examples use the input value of 16734, which is the SAS date value that corresponds to October 25, 2005.

<table>
<thead>
<tr>
<th>When the SAS Statement is ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>put date mmyy5.;</td>
<td>10M05</td>
</tr>
<tr>
<td>put date mmyy6.;</td>
<td>10M05</td>
</tr>
<tr>
<td>put date mmyy.;</td>
<td>10M2005</td>
</tr>
<tr>
<td>put date mmyy7.;</td>
<td>10M2005</td>
</tr>
<tr>
<td>put date mmyy10.;</td>
<td>10M2005</td>
</tr>
</tbody>
</table>

### See Also

Format:

“MMYY\(xw\). Format” on page 168

“YYMM\(w\). Format” on page 226

---

### MMYY\(xw\). Format

Writes date values in the form \(mm<yy>yy\) or \(mmX<yy>yy\), where \(X\) represents a specified separator and the year appears as either 2 or 4 digits.

**Category:** Date and Time

**Alignment:** right

### Syntax

\( \text{MMYY}\(xw\). \)
Syntax Description

\( x \)
identifies a separator or specifies that no separator appear between the month and
the year. Valid values for \( x \) are

- **C** separates with a colon
- **D** separates with a dash
- **N** indicates no separator
- **P** separates with a period
- **S** separates with a forward slash.

\( w \)
specifies the width of the output field.

**Default:** 7

**Range:** 5–32

**Interaction:** When \( x \) is set to N, no separator is specified. The width range is then
4–32, and the default changes to 6.

**Interaction:** When \( x \) has a value of C, D, P, or S and \( w \) has a value of 5 or 6, the
date appears with only the last two digits of the year. When \( w \) is 7 or more, the
date appears with a four-digit year.

**Interaction:** When \( x \) has a value of N and \( w \) has a value of 4 or 5, the date appears
with only the last two digits of the year. When \( x \) has a value of N and \( w \) is 6 or
more, the date appears with a four-digit year.

Details
The MMYY\( x \)w. format writes SAS date values in the form \( mm<yy>yy \) or \( mmX<yy>yy \),
where

- **\( mm \)** is an integer that represents the month.
- **X** is a specified separator.
- **\( <yy>yy \)** is a two-digit or four-digit integer that represents the year.

Examples
The following examples use the input value of 16631, which is the SAS date value
that corresponds to July14, 2005.

<table>
<thead>
<tr>
<th>When the SAS Statement is ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>put date mmyyc5.;</td>
<td>07:05</td>
</tr>
<tr>
<td>put date mmyyd.;</td>
<td>07-2005</td>
</tr>
</tbody>
</table>
When the SAS Statement is ... | The Result is ...
---|---
put date mmyyn4.; | 0705
put date mmyyp8.; | 07.2005
put date mmyys10.; | 07/2005

See Also

Format:

“MMYYw. Format” on page 167
“YYMMxw. Format” on page 227

MONNAME\textit{w.} Format

Writes date values as the name of the month

Category: Date and Time

Alignment: right

Syntax

MONNAME\textit{w.}

Syntax Description

\textit{w} specifies the width of the output field.

Default: 9

Range: 1–32

Tip: Use MONNAME3. to print the first three letters of the month name.

Details

If necessary, SAS truncates the name of the month to fit the format width.

Examples

The example table uses the input value of 16500, which is the SAS date value that corresponds to March 5, 2005.
MONTHw. Format

When the SAS Statement is ... The Result is ...

<table>
<thead>
<tr>
<th>put date monname1.;</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>put date monname3.;</td>
<td>Mar</td>
</tr>
<tr>
<td>put date monname5.;</td>
<td>March</td>
</tr>
</tbody>
</table>

See Also

Format:

“MONTHw. Format” on page 171

MONTHw. Format

Writes date values as the month of the year

Category: Date and Time

Alignment: right

Syntax

MONTHw.

Syntax Description

w

specifies the width of the output field.

Default: 2

Range: 1–21

Tip: Use MONTH1. to obtain a hex value.

Details

The MONTHw. format writes the month (1 through 12) of the year from a SAS date value.

Examples

The example table uses the input value of 15656, which is the SAS date value that corresponds to November 12, 2002.
When the SAS statement is ...

<table>
<thead>
<tr>
<th>The result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>put date month.;</td>
</tr>
</tbody>
</table>

### See Also

Format:

“MONNAMEw. Format” on page 170

---

### MONYYw. Format

**W**rites date values as the month and the year in the form *mmmyy* or *mmmyyyy*

**Category:** Date and Time  
**Alignment:** right

---

**Syntax**

**MONYYw.**

**Syntax Description**

\[ w \]

\( w \) specifies the width of the output field.

**Default:** 5

**Range:** 5–7

---

**Details**

The MONYYw. format writes SAS date values in the form *mmmyy* or *mmmyyyy*, where

\[ mmm \]

is the first three letters of the month name.

\[ yy \text{ or } yyyy \]

is a two-digit or four-digit integer that represents the year.

---

**Comparisons**

The MONYYw. format and the DTMONYYw. format are similar in that they both write date values. The difference is that MONYYw. expects a SAS date value as input, and DTMONYYw. expects a datetime value.
Examples

The example table uses the input value of 16794, which is the SAS date value that corresponds to December 24, 2005.

<table>
<thead>
<tr>
<th>When the SAS Statement is ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>put date monyy5.;</td>
<td>DEC05</td>
</tr>
<tr>
<td>put date monyy7.;</td>
<td>DEC2005</td>
</tr>
</tbody>
</table>

See Also

Formats:
“DTMONYYw. Format” on page 131
“DDMMYYw. Format” on page 123
“MMDDYYw. Format” on page 162
“YYMMDDw. Format” on page 229

Functions:
“MONTH Function” on page 655
“YEAR Function” on page 916

Informat:
“MONYYw. Informat” on page 995

NEGPARENw.d Format

Writes negative numeric values in parentheses

Category: Numeric
Alignment: right

Syntax

NEGPARENw.d

Syntax Description

w
specifies the width of the output field.
Default: 6
Range: 1–32
\(d\)

optionally specifies the number of digits to the right of the decimal point in the numeric value.

**Details**

The `NEGPARENw.d` format attempts to right align output values. If the input value is negative, `NEGPARENw.d` displays the output by enclosing the value in parentheses, if the field that you specify is wide enough. Otherwise, it uses a minus sign to represent the negative value. If the input value is non-negative, `NEGPARENw.d` displays the value with a leading and trailing blank to ensure proper column alignment. It reserves the last column for a right parenthesis even when the value is positive.

**Comparisons**

The `NEGPARENw.d` format is similar to the `COMMAw.d` format in that it separates every three digits of the value with a comma.

**Examples**

```plaintext
put @1 sales negparen6.;
```

<table>
<thead>
<tr>
<th>When sales = ...</th>
<th>The result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>1000</td>
<td>1,000</td>
</tr>
<tr>
<td>-200</td>
<td>(200)</td>
</tr>
<tr>
<td>-2000</td>
<td>-2,000</td>
</tr>
</tbody>
</table>

---

**NENGOw. Format**

Writes date values as Japanese dates in the form `e.yymmdd`

**Category:** Date and Time

**Alignment:** left

**See:** The NENGO format in *SAS National Language Support (NLS): User’s Guide*
NLDATEmw. Format

Converts a SAS date value to the date value of the specified locale and then writes the value in the format of date

Category: Date and Time


NLDATEMNmw. Format

Converts a SAS date value to the date value of the specified locale and then writes the date value in the format of name of month

Category: Date and Time


NLDATEmw. Format

Converts a SAS date value to the date value of the specified locale, and then writes the date value in the format of the date and the day of week

Category: Date and Time


NLDATENNmw. Format

Converts the SAS date value to the date value of the specified locale and then writes the date value in the format of the name of day of week

Category: Date and Time


NLDATMmw. Format

Converts a SAS datetime value to the datetime value of the specified locale and then writes the value in the format of datetime
NLDATMAPw. Format

Converts a SAS datetime value to the datetime value of the specified locale and then writes the value in the format of datetime with a.m. or p.m.

Category: Date and Time

NLDATMTMw. Format

Converts the time portion of a SAS datetime value to the time-of-day value of the specified locale and then writes the value in the format of time of day

Category: Date and Time

NLDATMWw. Format

Converts a SAS date value to a datetime value of the specified locale and then writes the value in the format of day of week and datetime

Category: Date and Time

NLMNYw.d Format

Writes the monetary format of the local expression in the specified locale using local currency

Category: Numeric

NLMNYIw.d Format

Writes the monetary format of the international expression in the specified locale
**NLNUMw.d Format**

Writes the numeric format of the local expression in the specified locale

**Category:** Numeric
**See:** The NLNUM format in *SAS National Language Support (NLS): User’s Guide*

---

**NLNUMIw.d Format**

Writes the numeric format of the international expression in the specified locale

**Category:** Numeric
**See:** The NLNUMI format in *SAS National Language Support (NLS): User’s Guide*

---

**NLPCTw.d Format**

Writes percentage data of the local expression in the specified locale

**Category:** Numeric
**See:** The NLPCT format in *SAS National Language Support (NLS): User’s Guide*

---

**NLPCTIw.d Format**

Writes percentage data of the international expression in the specified locale

**Category:** Numeric
**See:** The NLPCTI format in *SAS National Language Support (NLS): User’s Guide*

---

**NLTIMAPw. Format**

Converts a SAS time value to the time value of a specified locale and then writes the value in the format of a time value with a.m. or p.m.
**NLTIMEw. Format**

Converts a SAS time value to the time value of the specified locale and then writes the value in the format of time

Category: Date and Time

**NUMXw.d Format**

Writes numeric values with a comma in place of the decimal point

Category: Numeric
Alignment: right

**Syntax**

NUMXw.d

**Syntax Description**

\(w\)

specifies the width of the output field.

Default: 12
Range: 1–32

\(d\)

optionally specifies the number of digits to the right of the decimal point (comma) in the numeric value.

**Details**

The NUMXw.d format writes numeric values with a comma in place of the decimal point.

**Comparisons**

The NUMXw.d format is similar to the w.d format except that NUMXw.d writes numeric values with a comma in place of the decimal point.
Examples

\[
\text{put } x \text{ numx10.2;}
\]

| When \( x = ... \) | The Result is ...
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>896.48</td>
<td>896,48</td>
</tr>
<tr>
<td>64.89</td>
<td>64,89</td>
</tr>
<tr>
<td>3064.10</td>
<td>3064,10</td>
</tr>
</tbody>
</table>

See Also

Format:
“\(w.d\) Format” on page 214

Informat:
“NUMX\(w.d\) Informat” on page 1000

OCTAL\(w\). Format

Converts numeric values to octal representation

Category: Numeric
Alignment: left

Syntax

OCTAL\(w\).

Syntax Description

\(w\)

specifies the width of the output field.

Default: 3
Range: 1–24

Details

If necessary, the OCTAL\(w\) format converts numeric values to integers before displaying them in octal representation.
Comparisons

OCTALw. converts numeric values to octal representation. The $OCTALw.$ format converts character values to octal representation.

Examples

```
put x octal6.;
```

<table>
<thead>
<tr>
<th>When x = ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>3592</td>
<td>007010</td>
</tr>
</tbody>
</table>

PDw.d Format

Writes data in packed decimal format

Category: Numeric
Alignment: left
See: PDw.d Format in the documentation for your operating environment.

Syntax

PDw.d

Syntax Description

`w`

specifies the width of the output field. The w value specifies the number of bytes, not the number of digits. (In packed decimal data, each byte contains two digits.)

Default: 1
Range: 1–16

`d`

optionally specifies to multiply the number by $10^d$.

Details

Different operating environments store packed decimal values in different ways. However, the PDw.d format writes packed decimal values with consistent results if the values are created in the same kind of operating environment that you use to run SAS.

Comparisons

The following table compares packed decimal notation in several programming languages:
Language | Notation
---|---
SAS | PD4.
COBOL | COMP-3 PIC S9(7)
IBM 370 assembler | PL4
PL/I | FIXED DEC

Examples

```plaintext
y=put(x,pd4.);
put y $hex8.;
```

<table>
<thead>
<tr>
<th>When x = ...</th>
<th>The Result* is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>00000128</td>
</tr>
</tbody>
</table>

* The result is a hexadecimal representation of a binary number written in packed decimal format. Each byte occupies one column of the output field.

**PDJULGw. Format**

Writes packed Julian date values in the hexadecimal format \texttt{yyyydddf} for IBM

**Category:** Date and Time

**Syntax**

PDJULGw.

**Syntax Description**

\( w \)

specifies the width of the output field.

**Default:** 4

**Range:** 3-16

**Details**

The PDJULGw. format writes SAS date values in the form \texttt{yyyydddf}, where

\( \texttt{yyyy} \)

is the two-byte representation of the four-digit Gregorian year.
**PDJULIw. Format**

*Writes packed Julian date values in the hexadecimal format ccyyydddF for IBM*  

**Category:** Date and Time

**Syntax**

```
PDJULIw.
```
Syntax Description

\( w \)
specifies the width of the output field.

Default: 4
Range: 3-16

Details

The PDJUL\( w \). format writes SAS date values in the form \( ccyydddF \), where

\( cc \)
is the one-byte representation of a two-digit integer that represents the century.

\( yy \)
is the one-byte representation of a two-digit integer that represents the year. The PDJUL\( w \). format makes an adjustment for the century byte by subtracting 1900 from the 4-digit Gregorian year to produce the correct packed decimal \( ccyy \) representation. A year value of 1998 is stored in \( ccyy \) as 0098, and a year value of 2011 is stored as 0111.

\( ddd \)
is the one-and-a-half byte representation of the three-digit integer that corresponds to the Julian day of the year, 1–365 (or 1–366 for leap years).

\( F \)
is the half byte that contains all binary 1s, which assigns the value as positive.

*Note:* SAS interprets a two-digit year as belonging to the 100-year span that is defined by the YEARCUTOFF= system option.

Examples

<table>
<thead>
<tr>
<th>When the SAS Statement is ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>date = '17mar2005'd;</td>
<td>010705F</td>
</tr>
<tr>
<td>juldate = put(date,pdjuli4.);</td>
<td></td>
</tr>
<tr>
<td>put juldate $hex8.;</td>
<td></td>
</tr>
<tr>
<td>date = '31dec2003'd;</td>
<td>0103365F</td>
</tr>
<tr>
<td>juldate = put(date,pdjuli4.);</td>
<td></td>
</tr>
<tr>
<td>put juldate $hex8.;</td>
<td></td>
</tr>
</tbody>
</table>

See Also

Formats:

“PDJULG\( w \). Format” on page 181
“JULIAN\( w \). Format” on page 161
“JULDAY\( w \). Format” on page 160
PERCENTw.d Format

Writes numeric values as percentages

Category: Numeric
Alignment: right

Syntax

PERCENTw.d

Syntax Description

w
specifies the width of the output field.
Default: 6
Range: 4–32

d
optionally specifies the number of digits to the right of the decimal point in the numeric value.
Range: 0–31
Requirement: must be less than w

Details

The PERCENTw.d format multiplies values by 100, formats them the same as the BESTw.d format, and adds a percent sign (%) to the end of the formatted value, while it encloses negative values in parentheses. The PERCENTw.d format allows room for a percent sign and parentheses, even if the value is not negative.
Examples

```
put @10 gain percent10. ;
```

<table>
<thead>
<tr>
<th>When gain = ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>10%</td>
</tr>
<tr>
<td>1.2</td>
<td>120%</td>
</tr>
<tr>
<td>-0.05</td>
<td>( 5% )</td>
</tr>
</tbody>
</table>

**PIBw.d Format**

Writes positive integer binary (fixed-point) values

- **Category:** Numeric
- **Alignment:** Left
- **See:** PIBw.d Format in the documentation for your operating environment.

**Syntax**

```
PIBw.d
```

**Syntax Description**

- `w`
  - specifies the width of the output field.
  - **Default:** 1
  - **Range:** 1–8

- `d`
  - optionally specifies to multiply the number by 10^d.

**Details**

All values are treated as positive. PIBw.d writes positive integer binary values with consistent results if the values are created in the same type of operating environment that you use to run SAS.

*Note:* Different operating environments store integer binary values in different ways. This concept is called byte ordering. For a detailed discussion about byte ordering, see “Byte Ordering for Integer Binary Data on Big Endian and Little Endian Platforms” on page 75.
Comparisons

- Positive integer binary values are the same as integer binary values except that the sign bit is part of the value, which is always a positive integer. The PIBw.d format treats all values as positive and includes the sign bit as part of the value.
- The PIBw.d format with a width of 1 results in a value that corresponds to the binary equivalent of the contents of a byte. This is useful if your data contain values between hexadecimal 80 and hexadecimal FF, where the high-order bit can be misinterpreted as a negative sign.
- The PIBw.d format is the same as the IBw.d format except that PIBw.d treats all values as positive values.
- The IBw.d and PIBw.d formats are used to write native format integers. (Native format allows you to read and write values that are created in the same operating environment.) The IBRw.d and PIBRw.d formats are used to write little endian integers in any operating environment.

To view a table that shows the type of format to use with big endian and little endian integers, see Table 3.1 on page 76.

To view a table that compares integer binary notation in several programming languages, see Table 3.2 on page 76.

Examples

```plaintext
y=put(x,pib1.);
puy $hex2.;
```

| When x = ... | The Result is ...
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0c</td>
</tr>
</tbody>
</table>

* The result is a hexadecimal representation of a one-byte binary number written in positive integer binary format, which occupies one column of the output field.

See Also

Format:

“PIBRw.d Format” on page 186

PIBRw.d Format

Writes positive integer binary (fixed-point) values in Intel and DEC formats

Category: Numeric
Syntax

PIBR\(w.d\)

Syntax Description

\(w\)

specifies the width of the input field.

**Default:** 1

**Range:** 1–8

\(d\)

optionally specifies to multiply the number by 10\(^d\).

Details

All values are treated as positive. PIBR\(w.d\) writes positive integer binary values that have been generated by and for Intel and DEC operating environments. Use PIBR\(w.d\) to write positive integer binary data from Intel or DEC environments on other operating environments. The PIBR\(w.d\) format in SAS code allows for a portable implementation for writing the data in any operating environment.

**Note:** Different operating environments store positive integer binary values in different ways. This concept is called byte ordering. For a detailed discussion about byte ordering, see “Byte Ordering for Integer Binary Data on Big Endian and Little Endian Platforms” on page 75.

Comparisons

- Positive integer binary values are the same as integer binary values except that the sign bit is part of the value, which is always a positive integer. The PIBR\(w.d\) format treats all values as positive and includes the sign bit as part of the value.
- The PIBR\(w.d\) format with a width of 1 results in a value that corresponds to the binary equivalent of the contents of a byte. This is useful if your data contain values between hexadecimal 80 and hexadecimal FF, where the high-order bit can be misinterpreted as a negative sign.
- On Intel and DEC operating environments, the PIB\(w.d\) and PIBR\(w.d\) formats are equivalent.
- The IB\(w.d\) and PIB\(w.d\) formats are used to write native format integers. (Native format allows you to read and write values that are created in the same operating environment.) The IBR\(w.d\) and PIBR\(w.d\) formats are used to write little endian integers in any operating environment.

To view a table that shows the type of format to use with big endian and little endian integers, see Table 3.1 on page 76.

To view a table that compares integer binary notation in several programming languages, see Table 3.2 on page 76.

Examples

```sas
y=put(x,pibr2.);
put y $hex4.;
```
When $x = \ldots$ The Result* is...

| $128$ | $8000$ |

*The result is a hexadecimal representation of a two-byte binary number written in positive integer binary format, which occupies one column of the output field.

### See Also

Informat:

“PIB$w.d$ Informat” on page 1008

---

### PKw.d Format

Writes data in unsigned packed decimal format

**Category:** Numeric

**Alignment:** left

---

### Syntax

PK$w.d$

### Syntax Description

- $w$
  - specifies the width of the output field.
  - **Default:** 1
  - **Range:** 1–16

- $d$
  - optionally specifies to multiply the number by $10^d$.
  - **Requirement:** must be less than $w$

### Details

Each byte of unsigned packed decimal data contains two digits.

### Comparisons

The PK$w.d$ format is similar to the PD$w.d$ format except that PK$w.d$ does not write the sign in the low-order byte.
Examples

```plaintext
y=put(x,pk4.);
put y $hex8.;
```

<table>
<thead>
<tr>
<th>When x = ...</th>
<th>The Result* is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>00000128</td>
</tr>
</tbody>
</table>

* The result is a hexadecimal representation of a four-byte number written in packed decimal format. Each byte occupies one column of the output field.

PVALUEw.d Format

**Writes** $p$-values

**Category:** Numeric

**Alignment:** right

**Syntax**

PVALUE$w.d$

**Syntax Description**

$w$

specifies the width of the output field.

- **Default:** 6
- **Range:** 3–32

$d$

optionally specifies the number of digits to the right of the decimal point in the numeric value.

- **Default:** the minimum of 4 and $w-2$
- **Restriction:** must be less than $w$

**Comparisons**

The PVALUE$w.d$ format follows the rules for the $w.d$ format, except that

- if the value $x$ is such that $0 <= x < 10^d$, $x$ prints as “<.0...01” with $d-1$ zeros
- missing values print as “.” unless you specify a different character by using the MISSING= system option
Examples

```
put x pvalue6.4;
```

<table>
<thead>
<tr>
<th>When x = ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>.05</td>
<td>0.0500</td>
</tr>
<tr>
<td>0.000001</td>
<td>&lt;.0001</td>
</tr>
<tr>
<td>0</td>
<td>&lt;.0001</td>
</tr>
<tr>
<td>.0123456</td>
<td>0.0123</td>
</tr>
</tbody>
</table>

QTRw. Format

Writes date values as the quarter of the year

Category: Date and Time

Alignment: right

Syntax

```
QTRw.
```

Syntax Description

```
w
```

specifies the width of the output field.

Default: 1

Range: 1–32

Examples

The example table uses the input value of 16500, which is the SAS date value that corresponds to March 5, 2005.
See Also

Format:

“QTRRw. Format” on page 191

QTRRw. Format

Writes date values as the quarter of the year in Roman numerals

Category:  Date and Time

Alignment:  right

Syntax

QTRRw.

Syntax Description

\( w \)

specifies the width of the output field.

Default:  3

Range:  3–32

Examples

The example table uses the input value of 16694, which is the SAS date value that corresponds to September 15, 2005.

<table>
<thead>
<tr>
<th>When the SAS Statement is ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>put date qtrr.;</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>put date qtrr.;</td>
<td>III</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
See Also

Format:
“QTRw. Format” on page 190

---

RBw.d Format

Writes real binary data (floating-point) in real binary format

Category: Numeric
Alignment: left
See: RBw.d Format in the documentation for your operating environment.

Syntax

\texttt{RBw.d}

Syntax Description

\textit{w}

specifies the width of the output field.

\textbf{Default:} 4
\textbf{Range:} 2–8

\textit{d}

optionally specifies to multiply the number by \(10^d\).

Details

The \texttt{RBw.d} format writes numeric data in the same way that SAS stores them. Because it requires no data conversion, \texttt{RBw.d} is the most efficient method for writing data with SAS.

\textbf{Note:} Different operating environments store real binary values in different ways. However, \texttt{RBw.d} writes real binary values with consistent results in the same kind of operating environment that you use to run SAS.

\textbf{CAUTION:}

Using \texttt{RB4. to write real binary data on equipment that conforms to the IEEE standard for floating-point numbers results in a truncated eight-byte (double-precision) number rather than a true four-byte (single-precision) floating-point number.}
Comparisons

The following table compares the names of real binary notation in several programming languages:

<table>
<thead>
<tr>
<th>Language</th>
<th>4 Bytes</th>
<th>8 Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAS</td>
<td>RB4.</td>
<td>RB8.</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>REAL*4</td>
<td>REAL*8</td>
</tr>
<tr>
<td>C</td>
<td>float</td>
<td>double</td>
</tr>
<tr>
<td>COBOL</td>
<td>COMP-1</td>
<td>COMP-2</td>
</tr>
<tr>
<td>IBM 370 assembler</td>
<td>E</td>
<td>D</td>
</tr>
</tbody>
</table>

Examples

```plaintext
y=put(x,rb8.);
put y $hex16.;
```

When x = ... The Result* is ...

| 128  | 4280000000000000 |

* The result is a hexadecimal representation of an eight-byte real binary number as it looks on an IBM mainframe. Each byte occupies one column of the output field.

ROMANw. Format

**Writes numeric values as roman numerals**

**Category:** Numeric

**Alignment:** left

**Syntax**

ROMAN\textit{w}.

**Syntax Description**

\textit{w}

specifies the width of the output field.

**Default:** 6
Range: 2–32

Details
The ROMANw. format truncates a floating-point value to its integer component before the value is written.

Examples

```
put @5 year roman10.;
```

<table>
<thead>
<tr>
<th>When year = ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1998</td>
<td>MCMXCVIII</td>
</tr>
</tbody>
</table>

S370FFw.d Format

Writes native standard numeric data in IBM mainframe format

Category: Numeric

Syntax

```
S370FFw.d
```

Syntax Description

- **w**
  - specifies the width of the output field.
  - Default: 12
  - Range: 1–32

- **d**
  - optionally specifies the power of 10 by which to divide the value.
  - Range: 0–31

Details

The S370FFw.d format writes numeric data in IBM mainframe format (EBCDIC). The EBCDIC numeric values are represented with one byte per digit. If EBCDIC is the native format, S370FFw.d performs no conversion.

If a value is negative, an EBCDIC minus sign precedes the value. A missing value is represented as a single EBCDIC period.

Comparisons

On an EBCDIC system, S370FFw.d behaves like the w.d format.
On all other systems, S370FF\textit{w.d} performs the same role for numeric data that the \$EBCDIC\textit{w} format does for character data.

**Examples**

\begin{verbatim}
y=put(x,s370ff5.); put y $hex10.;
\end{verbatim}

<table>
<thead>
<tr>
<th>When (x = \ldots)</th>
<th>The Result* is (\ldots)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(12345)</td>
<td>(F1F2F3F4F5)</td>
</tr>
</tbody>
</table>

* The result is the hexadecimal representation for the integer.

**See Also**

Formats:
- \"$EBCDIC\textit{w} Format\" on page 96
- \"\textit{w.d} Format\" on page 214

---

**S370FiB\textit{w.d} Format**

W r i t e s i n t e g e r b i n a r y (f i x e d-point) v a l u e s, i n c l u d i n g n e g a t i v e v a l u e s, i n I B M m a i n f r a m e f o r m a t

\textbf{Category:} Numeric

\textbf{Alignment:} left

**Syntax**

\texttt{S370FiB\textit{w.d}}

**Syntax Description**

\(w\)

specifies the width of the output field.

- **Default:** 4
- **Range:** 1–8

\(d\)

optionally specifies to multiply the number by \(10^d\).
Details

The S370FIBw.d format writes integer binary (fixed-point) values that are stored in IBM mainframe format, including negative values that are represented in two's complement notation. S370FIBw.d writes integer binary values with consistent results if the values are created in the same type of operating environment that you use to run SAS.

Use S370FIBw.d to write integer binary data in IBM mainframe format from data that are created in other operating environments.

Note: Different operating environments store integer binary values in different ways. This concept is called byte ordering. For a detailed discussion about byte ordering, see “Byte Ordering for Integer Binary Data on Big Endian and Little Endian Platforms” on page 75.

Comparisons

- If you use SAS on an IBM mainframe, S370FIBw.d and IBw.d are identical.
- S370FPIBw.d, S370FIBUw.d, and S370FIBw.d are used to write big endian integers in any operating environment.

To view a table that shows the type of format to use with big endian and little endian integers, see Table 3.1 on page 76.

To view a table that compares integer binary notation in several programming languages, see Table 3.2 on page 76.

Examples

```sas
y=put(x,s370fib4.);
put y $hex8.;
```

<table>
<thead>
<tr>
<th>When x = ...</th>
<th>The Result* is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>000000080</td>
</tr>
</tbody>
</table>

* The result is a hexadecimal representation of a 4-byte integer binary number. Each byte occupies one column of the output field.

See Also

Formats:

“S370FIBUw.d Format” on page 196
“S370FPIBw.d Format” on page 200

S370FIBUw.d Format

Writes unsigned integer binary (fixed-point) values in IBM mainframe format
Category: Numeric
Alignment: left

Syntax

S370FIBUw.d

Syntax Description

\( w \)

specifies the width of the output field.
Default: 4
Range: 1–8

\( d \)

optionally specifies to multiply the number by 10^d.

Details

The S370FIBUw.d format writes unsigned integer binary (fixed-point) values that are stored in IBM mainframe format, including negative values that are represented in two's complement notation. Unsigned integer binary values are the same as integer binary values, except that all values are treated as positive. S370FIBUw.d writes integer binary values with consistent results if the values are created in the same type of operating environment that you use to run SAS.

Use S370FIBUw.d to write unsigned integer binary data in IBM mainframe format from data that are created in other operating environments.

Note: Different operating environments store integer binary values in different ways. This concept is called byte ordering. For a detailed discussion about byte ordering, see “Byte Ordering for Integer Binary Data on Big Endian and Little Endian Platforms” on page 75.

Comparisons

- The S370FIBUw.d format is equivalent to the COBOL notation PIC 9(n) BINARY, where \( n \) is the number of digits.
- The S370FIBUw.d format is the same as the S370FIBw.d format except that the S370FIBUw.d format always uses the absolute value instead of the signed value.
- The S370FPIBw.d format writes all negative numbers as FFs, while the S370FIBUw.d format writes the absolute value.
- S370FPIBw.d, S370FIBUw.d, and S370FIBw.d are used to write big endian integers in any operating environment.

To view a table that shows the type of format to use with big endian and little endian integers, see Table 3.1 on page 76.

To view a table that compares integer binary notation in several programming languages, see Table 3.2 on page 76.
Examples

```plaintext
y = put(x, s370fibul.);
put y $hex2.;
```

<table>
<thead>
<tr>
<th>When x = ...</th>
<th>The Result* is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>245</td>
<td>F5</td>
</tr>
<tr>
<td>-245</td>
<td>F5</td>
</tr>
</tbody>
</table>

* The result is a hexadecimal representation of a one-byte integer binary number. Each byte occupies one column of the output field.

See Also

Formats:
- “S370FIBw.d Format” on page 195
- “S370FPIBw.d Format” on page 200

S370FPDw.d Format

Writes packed decimal data in IBM mainframe format

Category: Numeric
Alignment: left

Syntax

S370FPDw.d

Syntax Description

\( w \)

specifies the width of the output field.

Default: 1
Range: 1–16

\( d \)

optionally specifies to multiply the number by \(10^d\).

Details

Use S370FPDw.d in other operating environments to write packed decimal data in the same format as on an IBM mainframe computer.
Comparisons

The following table shows the notation for equivalent packed decimal formats in several programming languages:

<table>
<thead>
<tr>
<th>Language</th>
<th>Packed Decimal Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAS</td>
<td>S370FPD4.</td>
</tr>
<tr>
<td>PL/I</td>
<td>FIXED DEC(7,0)</td>
</tr>
<tr>
<td>COBOL</td>
<td>COMP-3 PIC S9(7)</td>
</tr>
<tr>
<td>IBM 370 assembler</td>
<td>PL4</td>
</tr>
</tbody>
</table>

Examples

```plaintext
y=put(x,s370fpd4.);
put y $hex8.;
```

<table>
<thead>
<tr>
<th>When x = ...</th>
<th>The Result* is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>0000128C</td>
</tr>
</tbody>
</table>

* The result is a hexadecimal representation of a binary number written in packed decimal format. Each byte occupies one column of the output field.

S370FPDUw.d Format

Writes unsigned packed decimal data in IBM mainframe format

Category: Numeric
Alignment: left

Syntax

S370FPDUw.d

Syntax Description

w

specifies the width of the output field.

Default: 1
Range: 1–16
\[ d \]

optionally specifies to multiply the number by \(10^d\).

**Details**

Use S370FPDUw.d in other operating environments to write unsigned packed decimal data in the same format as on an IBM mainframe computer.

**Comparisons**

- The S370FPDUw.d format is similar to the S370FPDw.d format except that the S370FPDw.d format always uses the absolute value instead of the signed value.
- The S370FPDUw.d format is equivalent to the COBOL notation PIC 9(\(n\)) PACKED-DECIMAL, where the \(n\) value is the number of digits.

**Examples**

```plaintext
y=put(x,s370fpdu2.);
poly y $hex4.;
```

<table>
<thead>
<tr>
<th>When x = ...</th>
<th>The Result* is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>123F</td>
</tr>
<tr>
<td>-123</td>
<td>123F</td>
</tr>
</tbody>
</table>

* The result is a hexadecimal representation of a binary number written in packed decimal format. Each two hexadecimal digits correspond to one byte of binary data, and each byte corresponds to one column of the output field.

---

**S370FPIBw.d Format**

**Syntax**

\[ S370FPIBw.d \]

**Syntax Description**

\( w \)

specifies the width of the output field.

- **Default:** 4
- **Range:** 1–8
\(d\)

optionally specifies to multiply the number by 10^d.

**Details**

Positive integer binary values are the same as integer binary values, except that all values are treated as positive. `S370FPIBw.d` writes integer binary values with consistent results if the values are created in the same type of operating environment that you use to run SAS.

Use `S370FPIBw.d` to write positive integer binary data in IBM mainframe format from data that are created in other operating environments.

*Note:* Different operating environments store integer binary values in different ways. This concept is called byte ordering. For a detailed discussion about byte ordering, see “Byte Ordering for Integer Binary Data on Big Endian and Little Endian Platforms” on page 75.

**Comparisons**

- If you use SAS on an IBM mainframe, `S370FPIBw.d` and `PIBw.d` are identical.
- The `S370FPIBw.d` format is the same as the `S370FIBw.d` format except that the `S370FPIBw.d` format treats all values as positive values.
- `S370FPIBw.d`, `S370FIBUw.d`, and `S370FIBw.d` are used to write big endian integers in any operating environment.

To view a table that shows the type of format to use with big endian and little endian integers, see Table 3.1 on page 76.

To view a table that compares integer binary notation in several programming languages, see Table 3.2 on page 76.

**Examples**

```plaintext
y=put(x,s370fpib1.);
p= put y $hex2.;
```

When x = … The Result* is …

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>Oc</td>
</tr>
</tbody>
</table>

*The result is a hexadecimal representation of a one-byte binary number written in positive integer binary format, which occupies one column of the output field.

**See Also**

Formats:

“S370FIBw.d Format” on page 195

“S370FIBUw.d Format” on page 196
S370FRBw.d Format

Writes real binary (floating-point) data in IBM mainframe format

Category: Numeric
Alignment: left

Syntax

S370FRBw.d

Syntax Description

\(w\)

specifies the width of the output field.

Default: 6
Range: 2–8

\(d\)

optionally specifies to multiply the number by \(10^d\).

Details

A floating-point value consists of two parts: a mantissa that gives the value and an exponent that gives the value’s magnitude.

Use S370FRBw.d in other operating environments to write floating-point binary data in the same format as on an IBM mainframe computer.

Comparisons

The following table shows the notation for equivalent floating-point formats in several programming languages:

<table>
<thead>
<tr>
<th>Language</th>
<th>4 Bytes</th>
<th>8 Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL/I</td>
<td>FLOAT BIN(21)</td>
<td>FLOAT BIN(53)</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>REAL*4</td>
<td>REAL*8</td>
</tr>
<tr>
<td>COBOL</td>
<td>COMP-1</td>
<td>COMP-2</td>
</tr>
<tr>
<td>IBM 370 assembler</td>
<td>E</td>
<td>D</td>
</tr>
<tr>
<td>C</td>
<td>float</td>
<td>double</td>
</tr>
</tbody>
</table>
Examples

```plaintext
y=put(x,s370frb6.);
put y $hex8.;
```

<table>
<thead>
<tr>
<th>When x = ...</th>
<th>The Result* is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>42800000</td>
</tr>
<tr>
<td>-123</td>
<td>C2800000</td>
</tr>
</tbody>
</table>

* The result is a hexadecimal representation of a binary number in zoned decimal format on an IBM mainframe computer. Each two hexadecimal digits correspond to one byte of binary data, and each byte corresponds to one column of the output field.

---

**S370FZDw.d Format**

Writes zoned decimal data in IBM mainframe format

**Category:** Numeric  
**Alignment:** left

---

**Syntax**

```
S370FZDw.d
```

**Syntax Description**

- **w**
  - specifies the width of the output field.  
  - **Default:** 8  
  - **Range:** 1–32

- **d**
  - optionally specifies to multiply the number by $10^d$.

---

**Details**

Use `S370FZDw.d` in other operating environments to write zoned decimal data in the same format as on an IBM mainframe computer.
Comparisons

The following table shows the notation for equivalent zoned decimal formats in several programming languages:

<table>
<thead>
<tr>
<th>Language</th>
<th>Zoned Decimal Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAS</td>
<td>S370FZD3.</td>
</tr>
<tr>
<td>PL/I</td>
<td>PICTURE '99T'</td>
</tr>
<tr>
<td>COBOL</td>
<td>PIC S9(3) DISPLAY</td>
</tr>
<tr>
<td>assembler</td>
<td>ZL3</td>
</tr>
</tbody>
</table>

Examples

```plaintext
y=put(x,s370fzd3.);
put y $hex6.;
```

<table>
<thead>
<tr>
<th>When x = ...</th>
<th>The Result* is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>F1F2C3</td>
</tr>
<tr>
<td>-123</td>
<td>F1F2D3</td>
</tr>
</tbody>
</table>

* The result is a hexadecimal representation of a binary number in zoned decimal format on an IBM mainframe computer. Each two hexadecimal digits correspond to one byte of binary data, and each byte corresponds to one column of the output field.

S370FZDLw.d Format

Writes zoned decimal leading–sign data in IBM mainframe format

Category: Numeric
Alignment: left

Syntax

S370FZDLw.d

Syntax Description

w
specifies the width of the output field.
Default: 8  
Range: 1–32

\textit{d} optionally specifies to multiply the number by \(10^{d}\).

**Details**

Use \texttt{S370FZDLw.d} in other operating environments to write zoned decimal leading-sign data in the same format as on an IBM mainframe computer.

**Comparisons**

- The \texttt{S370FZDLw.d} format is similar to the \texttt{S370FZDw.d} format except that the \texttt{S370FZDLw.d} format displays the sign of the number in the first byte of the formatted output.
- The \texttt{S370FZDLw.d} format is equivalent to the COBOL notation \texttt{PIC S9(n) DISPLAY SIGN LEADING}, where the \texttt{n} value is the number of digits.

**Examples**

```plaintext
y=put(x,s370fzd13.);
put y $hex6.;
```

<table>
<thead>
<tr>
<th>When \texttt{x} = ...</th>
<th>The Result* is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>\texttt{C1F2F3}</td>
</tr>
<tr>
<td>-123</td>
<td>\texttt{D1F2F3}</td>
</tr>
</tbody>
</table>

* The result is a hexadecimal representation of a binary number in zoned decimal format on an IBM mainframe computer. Each two hexadecimal digits correspond to one byte of binary data, and each byte corresponds to one column of the output field.

---

**S370FZDSw.d Format**

Writes zoned decimal separate leading-sign data in IBM mainframe format

**Category:** Numeric  
**Alignment:** left

**Syntax**

\texttt{S370FZDSw.d}
Syntax Description

\[ w \]
specifies the width of the output field.

Default: 8
Range: 2–32

\[ d \]
optionally specifies to multiply the number by \( 10^d \).

Details

Use S370FZDSw.d in other operating environments to write zoned decimal separate leading-sign data in the same format as on an IBM mainframe computer.

Comparisons

- The S370FZDSw.d format is similar to the S370FZDLw.d format except that the S370FZDSw.d format does not embed the sign of the number in the zoned output.
- The S370FZDSw.d format is equivalent to the COBOL notation PIC S9(n) DISPLAY SIGN LEADING SEPARATE, where the \( n \) value is the number of digits.

Examples

\[
y = \text{put (x,s370fzds4.});
\]
\[
\text{put y $hex8.};
\]

<table>
<thead>
<tr>
<th>When x = ...</th>
<th>The Result* is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>4EF1F2F3</td>
</tr>
<tr>
<td>-123</td>
<td>60F1F2F3</td>
</tr>
</tbody>
</table>

* The result is a hexadecimal representation of a binary number in zoned decimal format on an IBM mainframe computer. Each two hexadecimal digits correspond to one byte of binary data, and each byte corresponds to one column of the output field.

S370FZDTw.d Format

Writes zoned decimal separate trailing-sign data in IBM mainframe format

Category: Numeric
Alignment: left

Syntax

S370FZDTw.d
**Syntax Description**

\( w \)
- specifies the width of the output field.
- **Default**: 8
- **Range**: 2–32

\( d \)
- optionally specifies to multiply the number by \(10^d\).

**Details**

Use S370FZDT\(w.d\) in other operating environments to write zoned decimal separate trailing-sign data in the same format as on an IBM mainframe computer.

**Comparisons**

- The S370FZDT\(w.d\) format is similar to the S370FZDS\(w.d\) format except that the S370FZDT\(w.d\) format displays the sign of the number at the end of the formatted output.
- The S370FZDT\(w.d\) format is equivalent to the COBOL notation PIC S9(\(n\)) DISPLAY SIGN TRAILING SEPARATE, where the \(n\) value is the number of digits.

**Examples**

```plaintext
y=put (x,s370fzdt4.); ;
put y $hex8.;
```

<table>
<thead>
<tr>
<th>When x = ...</th>
<th>The Result* is</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>F1F2F34E</td>
</tr>
<tr>
<td>-123</td>
<td>F1F2F360</td>
</tr>
</tbody>
</table>

* The result is a hexadecimal representation of a binary number in zoned decimal format on an IBM mainframe computer. Each two hexadecimal digits correspond to one byte of binary data, and each byte corresponds to one column of the output field.

---

**S370FZDU\(w.d\) Format**

- **Category**: Numeric
- **Alignment**: left

**Syntax**

S370FZDU\(w.d\)
**Syntax Description**

\( w \)

specifies the width of the output field.
- **Default:** 8
- **Range:** 1–32

\( d \)

optionally specifies to multiply the number by \(10^d\).

**Details**

Use S370FZDU\( w,d \) in other operating environments to write unsigned zoned decimal data in the same format as on an IBM mainframe computer.

**Comparisons**

- The S370FZDU\( w,d \) format is similar to the S370FZD\( w,d \) format except that the S370FZDU\( w,d \) format always uses the absolute value of the number.
- The S370FZDU\( w,d \) format is equivalent to the COBOL notation PIC 9(n) DISPLAY, where the \( n \) value is the number of digits.

**Examples**

\[
y=\text{put} \ (x,\text{s370fzdu}\ 3.); \\
\text{put} \ y \ \text{\$hex6.};
\]

<table>
<thead>
<tr>
<th>When ( x = \ldots )</th>
<th>The Result* is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>F1F2F3</td>
</tr>
<tr>
<td>-123</td>
<td>F1F2F3</td>
</tr>
</tbody>
</table>

* The result is a hexadecimal representation of a binary number in zoned decimal format on an IBM mainframe computer. Each pair of hexadecimal digits (such as F1) corresponds to one byte of binary data, and each byte corresponds to one column of the output field.

**SSNw. Format**

* **Writes Social Security numbers**
  - **Category:** Numeric
  - **Alignment:** none

**Syntax**

\( \text{SSN}w. \)
Syntax Description

\( w \)

specifies the width of the output field.

Default: 11

Restriction: \( w \) must be 11

Details

If the value is missing, SAS writes nine single periods with dashes between the third and fourth periods and between the fifth and sixth periods. If the value contains fewer than nine digits, SAS right aligns the value and pads it with zeros on the left. If the value has more than nine digits, SAS writes it as a missing value.

Examples

```
put id ssn11.;
```

When id = ... The Result is ...

| 263878439 | 263–87–8439 |

TIME\( w.d \) Format

Writes time values as hours, minutes, and seconds in the form \( hh:mm:ss.ss \)

Category: Date and Time

Alignment: right

Syntax

```
TIMEw.d
```

Syntax Description

\( w \)

specifies the width of the output field.

Default: 8

Range: 2–20
Tip: Make \( w \) large enough to produce the desired results. To obtain a complete time value with three decimal places, you must allow at least 12 spaces: 8 spaces to the left of the decimal point, 1 space for the decimal point itself, and 3 spaces for the decimal fraction of seconds.

\( d \)

optionally specifies the number of digits to the right of the decimal point in the seconds value.

**Requirement:** must be less than \( w \)

**Range:** 1–19

**Details**

The `TIMEw.d` format writes SAS time values in the form `hh:mm:ss.ss`, where

- \( hh \) is an integer.

  *Note:* If \( hh \) is a single digit, `TIMEw.d` places a leading blank before the digit. For example, the `TIMEw.d` format writes 9:00 instead of 09:00.

- \( mm \) is the number of minutes, ranging from 00 through 59.

- \( ss.ss \) is the number of seconds, ranging from 00 through 59, with the fraction of a second following the decimal point.

**Comparisons**

The `TIMEw.d` format is similar to the `HHMMw.d` format except that `TIMEw.d` includes seconds.

**Examples**

The example table uses the input value of 59083, which is the SAS time value that corresponds to 4:24:43 PM.

<table>
<thead>
<tr>
<th>When the SAS Statement is ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>put begin time.;</code></td>
<td>16:24:43</td>
</tr>
</tbody>
</table>

**See Also**

Formats:

“HHMMw.d Format” on page 153
“HOURw.d Format” on page 155
“MMSSw.d Format” on page 166

Functions:
“HOUR Function” on page 579
“MINUTE Function” on page 646
“SECOND Function” on page 820
“TIME Function” on page 856

Informat:
“TIMEw. Informat” on page 1037

TIMEAMPMw.d Format

Writes time values as hours, minutes, and seconds in the form hh:mm:ss.ss with AM or PM

Category: Date and Time
Alignment: right

Syntax
TIMEAMPMw.d

Syntax Description

w
specifies the width of the output field.
Default: 11
Range: 2–20

d
optionally specifies the number of digits to the right of the decimal point in the seconds value.
Requirement: must be less than w
Range: 1–19

Details
The TIMEAMPMw.d format writes SAS time values in the form hh:mm:ss.ss with AM or PM, where

hh
is an integer that represents the hour.

mm
is an integer that represents the minutes.

ss.ss
is the number of seconds to two decimal places.
Times greater than 23:59:59 PM appear as the next day.

Make \( w \) large enough to produce the desired results. To obtain a complete time value with three decimal places and AM or PM, you must allow at least 11 spaces (\( hh:mm:ss \) PM). If \( w \) is less than 5, SAS writes AM or PM only.

**Comparisons**

- The TIMEAMPMM\( w.d \) format is similar to the TIMEM\( w.d \) format except, that TIMEAMPMM\( w.d \) prints AM or PM at the end of the time.
- TIME\( w.d \) writes hours greater than 23:59:59 PM, and TIMEAMPM\( w.d \) does not.

**Examples**

The example table uses the input value of 59083, which is the SAS time value that corresponds to 4:24:43 PM.

<table>
<thead>
<tr>
<th>When the SAS Statement is ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>put begin timeampm3.;</td>
<td>PM</td>
</tr>
<tr>
<td>put begin timeampm5.;</td>
<td>4 PM</td>
</tr>
<tr>
<td>put begin timeampm7.;</td>
<td>4:24 PM</td>
</tr>
<tr>
<td>put begin timeampm11.;</td>
<td>4:24:43 PM</td>
</tr>
</tbody>
</table>

**See Also**

Format:
“TIME\( w.d \) Format” on page 209

---

**TOD\( w.d \) Format**

Writes the time portion of datetime values in the form \( hh:mm:ss.ss \)

**Category:** Date and Time

**Alignment:** right

**Syntax**

TOD\( w.d \)

**Syntax Description**

\( w \)

specifies the width of the output field.
Default: 8
Range: 2–20

d
optionally specifies the number of digits to the right of the decimal point in the seconds value.

Requirement: must be less than w
Range: 1–19

Details
The TODw.d format writes SAS datetime values in the form hh:mm:ss.ss, where

hh
is an integer that represents the hour.

mm
is an integer that represents the minutes.

ss.ss
is the number of seconds to two decimal places.

Examples
The example table uses the input value of 1472049623, which is the SAS datetime value that corresponds to August 24, 2006, at 2:20:23 PM.

<table>
<thead>
<tr>
<th>When the SAS Statement is ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>put begin tod9.2;</td>
<td>14:20:23</td>
</tr>
</tbody>
</table>

See Also
Formats:
“TIMEw.d Format” on page 209
“TIMEAMPMw.d Format” on page 211
Function:
“TIMEPART Function” on page 856
Informat:
“TIMEw. Informat” on page 1037

VAXRBw.d Format
Writes real binary (floating-point) data in VMS format
Syntax

VAXRB\(w.d\)

Syntax Description

\(w\)

specifies the width of the output field.

Default: 4
Range: 2-8

\(d\)

optionally specifies the power of 10 by which to divide the value.

Details

Use the VAXRB\(w.d\) format to write data in native VAX/VMS floating-point notation.

Comparisons

If you use SAS that is running under VAX/VMS, then the VAXRB\(w.d\) and the RB\(w.d\) formats are identical.

Example

```plaintext
x=1;
y=put(x,vaxrb8.);
put y=hex16.;
```

<table>
<thead>
<tr>
<th>When x=</th>
<th>The result* is</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8040000000000000</td>
</tr>
</tbody>
</table>

* The result is the hexadecimal representation for the integer.

\(w.d\) Format

Writes standard numeric data one digit per byte

Category: Numeric
Alignment: right
Alias: Fw.d
See: w.d Format in the documentation for your operating environment.

Syntax

w.d

Syntax Description

w specifies the width of the output field.
Range: 1–32
Tip: Allow enough space to write the value, the decimal point, and a minus sign, if necessary.

d optionally specifies the number of digits to the right of the decimal point in the numeric value.
Range: 0–31
Requirement: must be less than w
Tip: If d is 0 or you omit d, w.d writes the value without a decimal point.

Details

The w.d format rounds to the nearest number that fits in the output field. If w.d is too small, SAS may shift the decimal to the BESTw. format. The w.d format writes negative numbers with leading minus signs. In addition, w.d right aligns before writing and pads the output with leading blanks.

Comparisons

The Zw.d format is similar to the w.d format except that Zw.d pads right-aligned output with 0s instead of blanks.

Examples

   put @7 x 6.3;

<table>
<thead>
<tr>
<th>When x = ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>23.45</td>
<td>23.450</td>
</tr>
</tbody>
</table>
**WEEKDATEw. Format**

Writes date values as the day of the week and the date in the form *day-of-week, month-name dd, yy* (or *yyyy*).

**Category:** Date and Time

**Alignment:** right

**Syntax**

WEEKDATEw.

**Syntax Description**

*w*

specifies the width of the output field.

**Default:** 29

**Range:** 3–37

**Details**

The WEEKDATEw. format writes SAS date values in the form *day-of-week, month-name dd, yy* (or *yyyy*), where

**dd**

is an integer that represents the day of the month.

**yy or yyyy**

is a two-digit or four-digit integer that represents the year.

If *w* is too small to write the complete day of the week and month, SAS abbreviates as needed.

**Comparisons**

The WEEKDATEw. format is the same as the WEEKDATXw. format except that WEEKDATXw. prints *dd* before the month’s name.

**Examples**

The example table uses the input value of 16601 which is the SAS date value that corresponds to June 14, 2005.
When the SAS Statement is ... | The Result is ...
---+---+---+---+---
put date weekdate3.;       | Tue
put date weekdate9.;       | Tuesday
put date weekdate15.;      | Tue, Jun 14, 05
put date weekdate17.;      | Tue, Jun 14, 2005

See Also

Formats:
- “DATEw. Format” on page 118
- “DDMMYYw. Format” on page 123
- “MMDDYYw. Format” on page 162
- “TODw.d Format” on page 212
- “WEEKDATXw. Format” on page 217
- “YYMMDDw. Format” on page 229

Functions:
- “JULDATE Function” on page 613
- “MDY Function” on page 643
- “WEEKDAY Function” on page 916

Informats:
- “DATEw. Informat” on page 979
- “DDMMYYw. Informat” on page 982
- “MMDDYYw. Informat” on page 993
- “YYMMDDw. Informat” on page 1044

WEEKDATXw. Format

Writes date values as the day of the week and date in the form day-of-week, dd month-name yy (or yyyy)

Category: Date and Time
Alignment: right

Syntax

WEEKDATXw.
Syntax Description

\(w\)
specifies the width of the output field.

**Default:** 29  
**Range:** 3–37

Details

The WEEKDATX\(w\). format writes SAS date values in the form *day-of-week, dd month-name, yy (or yyyy)*, where

\(dd\)

is an integer that represents the day of the month.

\(yy\) or \(yyyy\)

is a two-digit or four-digit integer that represents the year.

If \(w\) is too small to write the complete day of the week and month, SAS abbreviates as needed.

Comparisons

The WEEKDATE\(w\). format is the same as the WEEKDATX\(w\). format except that WEEKDATE\(w\). prints \(dd\) after the month’s name.

The WEEKDATX\(w\). format is the same as the DTWKDATX\(w\). format, except that DTWKDATX\(w\). expects a datetime value as input.

Examples

The example table uses the input value of 16490 which is the SAS date value that corresponds to February 23, 2005.

<table>
<thead>
<tr>
<th>When the SAS statement is ...</th>
<th>The result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>put date weekdatx.;</td>
<td>Wednesday, 23 February 2005</td>
</tr>
</tbody>
</table>

See Also

Formats:

“DTWKDATX\(w\). Format” on page 132
“DATE\(w\). Format” on page 118
“DDMMYY\(w\). Format” on page 123
“MMDDYY\(w\). Format” on page 162
“TOD\(w.d\) Format” on page 212
“WEEKDATE\(w\). Format” on page 216
“YYMMDDw. Format” on page 229

Functions:
“JULDATE Function” on page 613
“MDY Function” on page 643
“WEEKDAY Function” on page 916

Informats:
“DATEw. Informat” on page 979
“DDMMYYw. Informat” on page 982
“MMDDYYw. Informat” on page 993
“YYMMDDw. Informat” on page 1044

WEEKDAYw. Format

WEEKDAYw. Format writes date values as the day of the week.

Category: Date and Time

Alignment: right

Syntax

WEEKDAYw.

Syntax Description

\( w \)

specifies the width of the output field.

Default: 1

Range: 1–32

Details

The WEEKDAYw. format writes a SAS date value as the day of the week (where 1=Sunday, 2=Monday, and so on).

Examples

The example table uses the input value of 16469, which is the SAS date value that corresponds to February 2, 2005.

<table>
<thead>
<tr>
<th>When the SAS Statement is ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>put date weekday.;</td>
<td>4</td>
</tr>
</tbody>
</table>

----+----1
See Also

Format:
“DOWNAMEw. Format” on page 129

---

**WEEKUw. Format**

Writes a week number in decimal format by using the U algorithm

Category: Date and Time


---

**WEEKVw. Format**

Writes a week number in decimal format by using the V algorithm

Category: Date and Time


---

**WEEKWw. Format**

Reads the format of the number-of-week value within the year and returns a SAS-date value using the W algorithm

Category: Date and Time


---

**WORDDATEw. Format**

Writes date values as the name of the month, the day, and the year in the form *month-name dd, yyyy*

Category: Date and Time

Alignment: right

**Syntax**

WORDDATEw.
Syntax Description

\( w \)

specifies the width of the output field.

Default: 18

Range: 3–32

Details

The FORMATW. format writes SAS date values in the form \textit{month-name dd, yyyy}, where

\( dd \)

is an integer that represents the day of the month.

\( yyyy \)

is a four-digit integer that represents the year.

If the width is too small to write the complete month, SAS abbreviates as necessary.

Comparisons

The FORMATW. format is the same as the FORMATXW. format except that FORMATXW. prints \textit{dd} before the month’s name.

Examples

The example table uses the input value of 16601, which is the SAS date value that corresponds to June 14, 2005.

<table>
<thead>
<tr>
<th>When the SAS Statement is ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>put term worddate3.;</td>
<td>Jun</td>
</tr>
<tr>
<td>put term worddate9.;</td>
<td>June</td>
</tr>
<tr>
<td>put term worddate12.;</td>
<td>Jun 14, 2005</td>
</tr>
<tr>
<td>put term worddate20.;</td>
<td>June 14, 2005</td>
</tr>
</tbody>
</table>

See Also

Format:

“FORMATXW. Format” on page 221

FORMATXW. Format

Writes date values as the day, the name of the month, and the year in the form \textit{dd month-name yyyy}
Category: Date and Time
Alignment: right

**Syntax**

WORDDATXw.

**Syntax Description**

\( w \)

specifies the width of the output field.

**Default:** 18  
**Range:** 3–32

**Details**

The WORDDATXw. format writes SAS date values in the form \( dd \) month-name, yyyy, where

\( dd \)

is an integer that represents the day of the month.

\( yyyy \)

is a four-digit integer that represents the year.

If the width is too small to write the complete month, SAS abbreviates as necessary.

**Comparisons**

The WORDDATXw. format is the same as the WORDDATEw. format except that WORDDATEw. prints \( dd \) after the month's name.

**Examples**

The example table uses the input value of 16500, which is the SAS date value that corresponds to March 5, 2005.

<table>
<thead>
<tr>
<th>When the SAS Statement is ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>put term worddatx.;</td>
<td>05 March 2005</td>
</tr>
</tbody>
</table>

**See Also**

Format:  
“WORDDATEw. Format” on page 220
WORDFw. Format

Writes numeric values as words with fractions that are shown numerically

Category:  Numeric
Alignment:  left

Syntax

WORDFw.

Syntax Description

\( w \)

specifies the width of the output field.

Default:  10
Range:  5–32767

Details

The WORDFw. format converts numeric values to their equivalent in English words, with fractions that are represented numerically in hundredths. For example, 8.2 prints as eight and 20/100.

Negative numbers are preceded by the word minus. When the value’s equivalent in words does not fit into the specified field, it is truncated on the right and the last character prints as an asterisk.

Comparisons

The WORDFw. format is similar to the WORDS\( w \). format except that WORDFw. prints fractions as numbers instead of words.

Examples

\[
\text{put price wordf15.;}
\]

<table>
<thead>
<tr>
<th>When price = ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5</td>
<td>two and 50/100</td>
</tr>
</tbody>
</table>
See Also

Format:
“WORDS\textsubscript{w}. Format” on page 224

\textbf{WORDS\textsubscript{w}. Format}

W\textsubscript{r}ites numeric values as words

\textbf{Category:} Numeric

\textbf{Alignment:} left

\textbf{Syntax}

\texttt{WORDS\textsubscript{w}.}

\textbf{Syntax Description}

\textit{w}

specifies the width of the output field.

\textbf{Default:} 10

\textbf{Range:} 5–32767

\textbf{Details}

You can use the \texttt{WORDS\textsubscript{w}.} format to print checks with the amount written out below the payee line.

Negative numbers are preceded by the word minus. If the number is not an integer, the fractional portion is represented as hundredths. For example, 5.3 prints as five and thirty hundredths. When the value's equivalent in words does not fit into the specified field, it is truncated on the right and the last character prints as an asterisk.

\textbf{Comparisons}

The \texttt{WORDS\textsubscript{w}.} format is similar to the \texttt{WORDF\textsubscript{w}.} format except that \texttt{WORDS\textsubscript{w}.} prints fractions as words instead of numbers.

\textbf{Examples}

\begin{verbatim}
put price words23.;
\end{verbatim}

\begin{tabular}{ll}
\textbf{When price = ...} & \textbf{The Result is ...} \\
\hline
\texttt{2.1} & \texttt{two and ten hundredths} \\
\end{tabular}
YEARw. Format

See Also

Format:
“WORDFw. Format” on page 223

YEARw. Format

Writes date values as the year

Category: Date and Time
Alignment: right

Syntax

YEARw.

Syntax Description

w
specifies the width of the output field.
Default: 4
Range: 2–32
Tip: If w is less than 4, the last two digits of the year print; otherwise, the year value prints as four digits.

Comparisons

The YEARw. format is similar to the DTYEARw. format in that they both write date values. The difference is that YEARw. expects a SAS date value as input, and DTYEARw. expects a datetime value.

Examples

The example table uses the input value of 16601, which is the SAS date value that corresponds to June 14, 2005.

<table>
<thead>
<tr>
<th>When the SAS Statement is ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>put date year2.;</td>
<td>05</td>
</tr>
<tr>
<td>put date year4.;</td>
<td>2005</td>
</tr>
</tbody>
</table>
See Also

Format:
“DTYEARw. Format” on page 133

YENw.d Format

Writes numeric values with yen signs, commas, and decimal points

Category: Numeric
Alignment: right

YYMMw. Format

Writes date values in the form <yy>yyMmm, where M is the separator and the year appears as either 2 or 4 digits

Category: Date and Time
Alignment: right

Syntax
YYMMw.

Syntax Description

w
specifies the width of the output field.
Default: 7
Range: 5–32
Interaction: When w has a value of 5 or 6, the date appears with only the last two digits of the year. When w is 7 or more, the date appears with a four-digit year.

Details
The YYMMw. format writes SAS date values in the form <yy>yyMmm, where

<yy>yy
is a two-digit or four-digit integer that represents the year.

M
is the character separator.
is an integer that represents the month.

Examples

The following examples use the input value of 16734, which is the SAS date value that corresponds to October 25, 2005.

<table>
<thead>
<tr>
<th>When the SAS Statement is ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>put date yymm5.;</td>
<td>05M10</td>
</tr>
<tr>
<td>put date yymm6.;</td>
<td>05M10</td>
</tr>
<tr>
<td>put date yymm.;</td>
<td>2005M10</td>
</tr>
<tr>
<td>put date yymm7.;</td>
<td>2005M10</td>
</tr>
<tr>
<td>put date yymm10.;</td>
<td>2005M10</td>
</tr>
</tbody>
</table>

See Also

Format:

“MMYYwx. Format” on page 167
“YYMMxw. Format” on page 227

YYMMwx. Format

 Writes date values in the form `<yy>yyymm or <yy>yyXmm, where X represents a specified separator and the year appears as either 2 or 4 digits.

 Category: Date and Time
 Alignment: right

Syntax

YYMMwx.

Syntax Description

x

identifies a separator or specifies that no separator appear between the year and the month. Valid values for x are:

C

separates with a colon
D
  separates with a dash

N
  indicates no separator

P
  separates with a period

S
  separates with a forward slash.

\( w \)

specifies the width of the output field.

Default:  \( 7 \)

Range:  \( 5–32 \)

Interaction:  When \( x \) is set to \( N \), no separator is specified. The width range is then 4–32, and the default changes to 6.

Interaction:  When \( x \) has a value of \( C \), \( D \), \( P \), or \( S \) and \( w \) has a value of 5 or 6, the date appears with only the last two digits of the year. When \( w \) is 7 or more, the date appears with a four-digit year.

Interaction:  When \( x \) has a value of \( N \) and \( w \) has a value of 4 or 5, the date appears with only the last two digits of the year. When \( x \) has a value of \( N \) and \( w \) is 6 or more, the date appears with a four-digit year.

Details

The \( YYMMxw. \) format writes SAS date values in the form \(<yy>yy\text{mm} \) or \(<yy>yyX\text{mm} \), where

\(<yy>yy\)

is a two-digit or four-digit integer that represents the year.

\(X\)

is a specified separator.

\(mm\)

is an integer that represents the month.

Examples

The following examples use the input value of 16631, which is the SAS date value that corresponds to July 14, 2005.

<table>
<thead>
<tr>
<th>When the SAS Statement is ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>put date yymmc5.;</td>
<td>05:07</td>
</tr>
<tr>
<td>put date yymmd.;</td>
<td>2005-07</td>
</tr>
<tr>
<td>put date yymmn4.;</td>
<td>0507</td>
</tr>
<tr>
<td>put date yymmp8.;</td>
<td>2005.07</td>
</tr>
<tr>
<td>put date yymms10.;</td>
<td>2005/07</td>
</tr>
</tbody>
</table>
See Also

Format:
“MMYYwxw. Format” on page 168
“YYMMwx. Format” on page 226

YYMMDDwx. Format

Writes date values in the form <yy>yyymmd or <yy>yy-mm-dd, where a dash is the separator and the year appears as either 2 or 4 digits

Category: Date and Time
Alignment: right

Syntax
YYMMDDwx.

Syntax Description

w

specifies the width of the output field.
Default: 8
Range: 2–10

Interaction: When w has a value of from 2 to 5, the date appears with as much of the year and the month as possible. When w is 7, the date appears as a two-digit year without dashes.

Details

The YYMMDDwx. format writes SAS date values in the form <yy>yyymmd or <yy>yy–mm–dd, where

<yy>yy

is a two-digit or four-digit integer that represents the year.

–

is the separator.

mm

is an integer that represents the month.

dd

is an integer that represents the day of the month.

Examples

The following examples use the input value of 16529, which is the SAS date value that corresponds to April 3, 2005.
### YYMMDDxw. Format

**Writes date values in the form** `<yy>yyymmdd` or `<yy>yyXmmXdd`, where X represents a specified separator and the year appears as either 2 or 4 digits

**Category:** Date and Time  
**Alignment:** right

#### Syntax

YYMMDDxw.
**Syntax Description**

$x$

identifies a separator or specifies that no separator appear between the year, the month, and the day. Valid values for $x$ are:

- **B**: separates with a blank
- **C**: separates with a colon
- **D**: separates with a dash
- **N**: indicates no separator
- **P**: separates with a period
- **S**: separates with a slash.

$w$

specifies the width of the output field.

**Default:** 8

**Range:** 2–10

**Interaction:** When $w$ has a value of from 2 to 5, the date appears with as much of the year and the month. When $w$ is 7, the date appears as a two-digit year without separators.

**Interaction:** When $x$ has a value of N, the width range is 2–8.

**Details**

The YYMMD$tw$ format writes SAS date values in the form $<yy>yyyymmdd$ or $<yy>yyXmmXdd$, where

- $<yy>yy$ is a two-digit or four-digit integer that represents the year.
- $X$ is a specified separator.
- $mm$ is an integer that represents the month.
- $dd$ is an integer that represents the day of the month.

**Examples**

The following examples use the input value of 16731, which is the SAS date value that corresponds to October 22, 2005.
When the SAS Statement is ... | The Result is ...  
---+---+----
put day yymmddc5.; | 05:10
put day yymmdd8.; | 05-10-22
put day yymmddp10.; | 2005.10.22
put day yymmddn8.; | 20051022

See Also

Formats:
“DATEw. Format” on page 118
“DDMMYYxw. Format” on page 124
“MMDDYYxw. Format” on page 164
“YYMMDDw. Format” on page 229

Functions:
“DAY Function” on page 481
“MDY Function” on page 643
“MONTH Function” on page 655
“YEAR Function” on page 916

Informat:
“YYMMDDw. Informat” on page 1044

YYMONw. Format

 Writes date values in the form yymm or yyyyymm  

Category: Date and Time  
Alignment: right

Syntax

YYMONw.

Syntax Description

\( w \)
specifies the width of the output field. If the format width is too small to print a four-digit year, only the last two digits of the year are printed.

Default: 7  
Range: 5–32
Details

The YYMONw. format abbreviates the month’s name to three characters.

Examples

The example table uses the input value of 16601, which is the SAS date value that corresponds to June 14, 2005.

<table>
<thead>
<tr>
<th>When the SAS Statement is ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>put date yymon6.;</td>
<td>05JUN</td>
</tr>
<tr>
<td>put date yymon7.;</td>
<td>2005JUN</td>
</tr>
</tbody>
</table>

See Also

Format:

“MMYYw. Format” on page 167

YYQw. Format

Writes date values in the form <yy>yyQq, where Q is the separator, the year appears as either 2 or 4 digits, and q is the quarter of the year

Category:  Date and Time
Alignment:  right

Syntax

YYQw.

Syntax Description

w

specifies the width of the output field.

Default:  6

Range:  4–32

Interaction:  When w has a value of 4 or 5, the date appears with only the last two digits of the year. When w is 6 or more, the date appears with a four-digit year.
Details

The YYQw. format writes SAS date values in the form <yy>yyQq, where

<yy>yy
is a two-digit or four-digit integer that represents the year.

Q
is the character separator.

q
is an integer (1,2,3, or 4) that represents the quarter of the year.

Examples

The following examples use the input value of 16601, which is the SAS date value that corresponds to June 14, 2005.

<table>
<thead>
<tr>
<th>When the SAS Statement is ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>put date yyq4.;</td>
<td>05Q2</td>
</tr>
<tr>
<td>put date yyq5.;</td>
<td>05Q2</td>
</tr>
<tr>
<td>put date yyq.;</td>
<td>2005Q2</td>
</tr>
<tr>
<td>put date yyq6.;</td>
<td>2005Q2</td>
</tr>
<tr>
<td>put date yyq10.;</td>
<td>2005Q2</td>
</tr>
</tbody>
</table>

See Also

Formats:

“YYQxw. Format” on page 234
“YYQRw. Format” on page 236

YYQxw. Format

Writes date values in the form <yy>yyq or <yy>yyXq, where X represents a specified separator, the year appears as either 2 or 4 digits, and q is the quarter of the year

Category: Date and Time
Alignment: right

Syntax

YYQxw.
Syntax Description

\( x \)

identifies a separator or specifies that no separator appear between the year and the quarter. Valid values for \( x \) are:

C
  separates with a colon

D
  separates with a dash

N
  indicates no separator

P
  separates with a period

S
  separates with a forward slash.

\( w \)

specifies the width of the output field.

**Default:** 6

**Range:** 4–32

**Interaction:** When \( x \) is set to \( N \), no separator is specified. The width range is then 3–32, and the default changes to 5.

**Interaction:** When \( w \) has a value of 4 or 5, the date appears with only the last two digits of the year. When \( w \) is 6 or more, the date appears with a four-digit year.

**Interaction:** When \( x \) has a value of \( N \) and \( w \) has a value of 3 or 4, the date appears with only the last two digits of the year. When \( x \) has a value of \( N \) and \( w \) is 5 or more, the date appears with a four-digit year.

Details

The `YYQwx.` format writes SAS date values in the form `<yy>yyq` or `<yy>yyXq`, where

`<yy>yy`

is a two-digit or four-digit integer that represents the year.

`X`

is a specified separator.

`q`

is an integer (1,2,3, or 4) that represents the quarter of the year.

Examples

The following examples use the input value of 16631, which is the SAS date value that corresponds to July 14, 2005.

<table>
<thead>
<tr>
<th>When the SAS Statement is ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>put date yyqc4.;</td>
<td>05:3</td>
</tr>
<tr>
<td>put date yyqd.;</td>
<td>2005–3</td>
</tr>
</tbody>
</table>
### YYQRw. Format

Writes date values in the form `<yy>`yyQ<sub>qr</sub>, where Q is the separator, the year appears as either 2 or 4 digits, and <sub>qr</sub> is the quarter of the year expressed in roman numerals.

**Category:** Date and Time  
**Alignment:** right

### Syntax Description

**w**  
Specifies the width of the output field.  
**Default:** 8  
**Range:** 6–32  
**Interaction:** When the value of <sub>w</sub> is too small to write a four-digit year, the date appears with only the last two digits of the year.

### Details

The YYQR<sub>w</sub>. format writes SAS date values in the form `<yy>`yyQ<sub>qr</sub>, where

- `<yy>`yy  
  is a two-digit or four-digit integer that represents the year.
- Q  
  is the character separator.
- `<sub>qr</sub>`  
  is a Roman numeral (I, II, III, or IV) that represents the quarter of the year.

---

### See Also

Formats:  
“YYQ<sub>w</sub>. Format” on page 233  
“YYQRx<sub>w</sub>. Format” on page 237

---

<table>
<thead>
<tr>
<th>When the SAS Statement is ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>put date yyqa3.;</td>
<td>053</td>
</tr>
<tr>
<td>put date yyqsp6.;</td>
<td>2005.3</td>
</tr>
<tr>
<td>put date yyqs8.;</td>
<td>2005/3</td>
</tr>
</tbody>
</table>
Examples

The following examples use the input value of 16601, which is the SAS date value that corresponds to June 14, 2005.

<table>
<thead>
<tr>
<th>When the SAS Statement is ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>put date yyqr6.;</td>
<td>05QII</td>
</tr>
<tr>
<td>put date yyqr7.;</td>
<td>2005QII</td>
</tr>
<tr>
<td>put date yyqr.;</td>
<td>2005QII</td>
</tr>
<tr>
<td>put date yyqr8.;</td>
<td>2005QII</td>
</tr>
<tr>
<td>put date yyqr10.;</td>
<td>2005QII</td>
</tr>
</tbody>
</table>

See Also

Format:

“YYQw. Format” on page 233
“YYQRxw. Format” on page 237

YYQRxw. Format

Writes date values in the form <yy>yyqr or <yy>yyXqr, where X represents a specified separator, the year appears as either 2 or 4 digits, and qr is the quarter of the year expressed in Roman numerals.

Category: Date and Time
Alignment: right

Syntax

YYQRxw.

Syntax Description

x identifies a separator or specifies that no separator appear between the year and the quarter. Valid values for x are:

C separates with a colon

D separates with a dash
N  indicates no separator
P  separates with a period
S  separates with a forward slash.

\(w\)
specifies the width of the output field.

**Default:**  8  
**Range:**  6–32  
**Interaction:** When \(x\) is set to \(N\), no separator is specified. The width range is then 5–32, and the default changes to 7.

**Interaction:** When the value of \(w\) is too small to write a four-digit year, the date appears with only the last two digits of the year.

**Details**

The \(YYQRxw\). format writes SAS date values in the form \(<yy>yyqr\) or \(<yy>yyXqr\), where

\(<yy>yy\)
is a two-digit or four-digit integer that represents the year.

\(X\)
is a specified separator.

\(qr\)
is a Roman numeral (I, II, III, or IV) that represents the quarter of the year.

**Examples**

The following examples use the input value of 16431, which is the SAS date value that corresponds to December 24, 2004.

| When the SAS Statement is       | The Result is ...
|---------------------------------|-----------------
| put date yyqrc6.;               | 04:IV           |
| put date yyqrd.;                | 2004-IV         |
| put date yyqnn5.;               | 04IV            |
| put date yyqrp8.;               | 2004.IV         |
| put date yyqrs10.;              | 2004/IV         |

**See Also**

Format:

“\(YYxw\). Format” on page 234

“\(YYRw\). Format” on page 236
Zw.d Format

Writes standard numeric data with leading 0s

Category: Numeric
Alignment: right

Syntax

Zw.d

Syntax Description

\(w\)

specifies the width of the output field.

Default: 1
Range: 1–32
Tip: Allow enough space to write the value, the decimal point, and a minus sign, if necessary.

\(d\)

optionally specifies the number of digits to the right of the decimal point in the numeric value.

Range: 0–31
Tip: If \(d\) is 0 or you omit \(d\), Zw.d writes the value without a decimal point.

Details

The Zw.d format writes standard numeric values one digit per byte and fills in 0s to the left of the data value.

The Zw.d format rounds to the nearest number that will fit in the output field. If \(w.d\) is too large to fit, SAS might shift the decimal to the BESTw format. The Zw.d format writes negative numbers with leading minus signs. In addition, it right aligns before writing and pads the output with leading zeros.

Comparisons

The Zw.d format is similar to the w.d format except that Zw.d pads right-aligned output with 0s instead of blanks.

Examples

```
put @5 seqnum z8.;
```

<table>
<thead>
<tr>
<th>When seqnum = ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1350</td>
<td>00001350</td>
</tr>
</tbody>
</table>
ZD\textit{w.d} Format

\textbf{ZD\textit{w.d} Format}

\begin{itemize}
\item \textbf{Writing numeric data in zoned decimal format}
\item \textbf{Category}: Numeric
\item \textbf{Alignment}: left
\item \textbf{See}: ZD\textit{w.d} Format in the documentation for your operating environment.
\end{itemize}

\section*{Syntax}

\textit{ZDw.d}

\section*{Syntax Description}

\textit{w}

- specifies the width of the output field.
- \textbf{Default}: 1
- \textbf{Range}: 1–32

\textit{d}

- optionally specifies to multiply the number by \(10^d\).

\section*{Details}

The zoned decimal format is similar to standard numeric format in that every digit requires one byte. However, the value's sign is in the last byte, along with the last digit.

\textit{Note}: Different operating environments store zoned decimal values in different ways. However, the ZD\textit{w.d} format writes zoned decimal values with consistent results if the values are created in the same kind of operating environment that you use to run SAS. △

\section*{Comparisons}

The following table compares the zoned decimal format with notation in several programming languages:

\begin{tabular}{|l|l|}
\hline
\textbf{Language} & \textbf{Zoned Decimal Notation} \\
\hline
SAS & ZD3. \\
PL/1 & PICTURE '99T' \\
COBOL & DISPLAY PIC S 999 \\
IBM 370 assembler & ZL3 \\
\hline
\end{tabular}
Examples

```y=put(x,zd4.); put y $hex8.;```

When x = ... The Result* is ...

| 120 | 0F01F2C0 |

* The result is a hexadecimal representation of a binary number in zoned decimal format on an IBM mainframe computer. Each byte occupies one column of the output field.
Functions and CALL Routines

Definitions of Functions and CALL Routines

Definition of Functions
Definition of CALL Routines

Syntax

Syntax of Functions
Syntax of CALL Routines

Using Functions

Restrictions on Function Arguments
Notes on Descriptive Statistic Functions
Notes on Financial Functions
Special Considerations for Depreciation Functions
Using DATA Step Functions within Macro Functions
Using Functions to Manipulate Files

Using Random-Number Functions and CALL Routines

Seed Values
Comparison of Random-Number Functions and CALL Routines
Examples

Example 1: Generating Multiple Streams from a CALL Routine
Example 2: Assigning Values from a Single Stream to Multiple Variables

Pattern Matching Using SAS Regular Expressions (RX) and Perl Regular Expressions (PRX)

Definition of Pattern Matching and Regular Expressions
Definition of SAS Regular Expression (RX) Functions and CALL Routines
Definition of Perl Regular Expression (PRX) Functions and CALL Routines
Benefits of Using Perl Regular Expressions in the DATA Step
Using Perl Regular Expressions in the DATA Step

License Agreement

Syntax of Perl Regular Expressions
Example 1: Validating Data
Example 2: Replacing Text
Example 3: Extracting a Substring from a String

Writing Perl Debug Output to the SAS Log

Base SAS Functions for Web Applications

Functions and CALL Routines by Category

Dictionary

ABS Function
ADDR Function
ADDRLONG Function
AIRY Function
ANYALNUM Function
ANYALPHA Function
ANYCNTRL Function
CALL RANUNI Routine 383
CALL RXCHANGE Routine 385
CALL RXFREE Routine 386
CALL RXSUBSTR Routine 387
CALL SCAN Routine 388
CALL SCANQ Routine 390
CALL SET Routine 392
CALL SLEEP Routine 394
CALL SOFTMAX Routine 395
CALL STDIZE Routine 396
CALL STREAMINIT Routine 400
CALL SYMPUT Routine 401
CALL SYMPUTX Routine 402
CALL SYSTEM Routine 403
CALL TANH Routine 404
CALL VNAME Routine 405
CALL VNEXT Routine 406
CAT Function 408
CATS Function 410
CATT Function 412
CATX Function 414
CDF Function 416
CEIL Function 429
CEILZ Function 430
CEXIST Function 431
CHOOSEC Function 432
CHOOSEN Function 433
CINV Function 434
CLOSE Function 436
CNONCT Function 437
COALESCE Function 438
COALESCEC Function 439
COLLATE Function 440
COMB Function 442
COMPARE Function 443
COMPBL Function 446
COMPGED Function 447
COMPLEV Function 452
COMPOUND Function 455
COMPRESS Function 456
CONSTANT Function 459
CONVX Function 462
CONVXP Function 463
COS Function 465
COSH Function 465
COUNT Function 466
COUNTC Function 468
CSS Function 469
CUROBS Function 470
CV Function 471
DACCDB Function 472
DACCDBSL Function 473
DACCSL Function 474
DACCSYD Function 474
DACCTAB Function 475
DAIRY Function 476
DATDIF Function 477
DATE Function 478
DATEJUL Function 479
DATEPART Function 479
DATETIME Function 480
DAY Function 481
DCLOSE Function 481
DCREATE Function 483
DEPDB Function 484
DEPDBSL Function 485
DEPSL Function 486
DEPSYD Function 487
DEPTAB Function 488
DEQUOTE Function 489
DEVIANC Function 491
DHMS Function 495
DIF Function 496
DIGAMMA Function 497
DIM Function 498
DINFO Function 499
DNUM Function 501
DOPEN Function 502
DOPTNAME Function 503
DOPTNUM Function 505
DREAD Function 506
DROPNOTE Function 507
DSNAME Function 508
DUR Function 509
DURP Function 510
ERF Function 511
ERFC Function 512
EUROCURR Function 513
EXIST Function 513
EXP Function 515
FACT Function 516
FAPPEND Function 517
FCLOSE Function 518
FCOL Function 520
FDELETE Function 521
FETCH Function 522
FETCHOBS Function 523
FEXIST Function 525
FGET Function 526
FILEEXIST Function 527
FILENAME Function 528
FILEREF Function 531
FIND Function 532
FINDC Function 534
FINFO Function 538
FINV Function 539
FIPNAME Function 540
FIPNAMEL Function 541
FIPSTATE Function 542
FLOOR Function 544
FLOORZ Function 545
FNONCT Function 546
FNONE Function 547
FOPEN Function 549
FOPNAME Function 551
FOPNUM Function 553
FPOINT Function 554
FPOS Function 555
FPUT Function 556
FREAD Function 558
FREWIND Function 559
FRLEN Function 560
FSEP Function 561
FUZZ Function 562
FWRITE Function 563
GAMINV Function 565
GAMMA Function 565
GEOMEAN Function 566
GEOMEANZ Function 568
GETOPTION Function 569
GETVAN Function 571
GETVARP Function 572
HARMEAN Function 574
HARMEANZ Function 575
HBOUND Function 576
HMS Function 578
HOUR Function 579
HTMLDECODE Function 580
HTMLINCODE Function 581
IBESSEL Function 583
IFC Function 583
IFN Function 585
INDEX Function 588
INDEXC Function 589
INDEXW Function 590
INPUT Function 592
INPUTC Function 594
INPUTN Function 596
INT Function 597
INTCK Function 598
INTNX Function 602
INTTR Function 607
INTZ Function 608
IORCMSG Function 610
IQR Function 611
IRR Function 612
JBESSEL Function 613
JULDATE Function 613
JULDATE7 Function 614
KCOMPARE Function 615
KCOMPRESSION Function 615
KCOUNT Function 616
MOPEN Function 656
MORT Function 658
N Function 659
NETPV Function 660
NLDATE Function 661
NLDATM Function 661
NLTIME Function 661
NLITERAL Function 662
NMISS Function 664
NORMAL Function 664
NOTALNUM Function 665
NOTALPHA Function 666
NOTCNTRL Function 668
NOTDIGIT Function 669
NOTE Function 671
NOTFIRST Function 672
NOTGRAPH Function 674
NOTLOWER Function 676
NOTNAME Function 677
NOTPRINT Function 679
NOTPUNCT Function 680
NOTSPACE Function 682
NOTUPPER Function 684
NOTXDIGIT Function 685
NPV Function 687
NVALID Function 688
OPEN Function 690
ORDINAL Function 692
PATHNAME Function 692
PCTL Function 694
PDF Function 695
PEEK Function 708
PEEKC Function 709
PEEKCLONG Function 712
PEEKLONG Function 713
PERM Function 714
POINT Function 715
POISSON Function 716
PROBBETA Function 717
PROBBNML Function 718
PROBBNRM Function 719
PROBCHI Function 720
PROBF Function 721
PROBGAM Function 722
PROBHYP Function 723
PROBIT Function 724
PROBMC Function 724
PROBNMG Function 735
PROBNRM Function 736
PROBT Function 736
PROPCASE Function 737
PRXCHANGE Function 739
PRXRANCE Function 743
PRXPEND Function 747
PRXPARSE Function 749
PRXPOSN Function 751
PRTRONGADD Function 754
PUT Function 754
PUTC Function 756
PUTN Function 758
PVP Function 759
QTR Function 760
QUANTILE Function 761
QUOTE Function 763
RANBIN Function 764
RANCAU Function 765
RAND Function 766
RANEXP Function 777
RANGAM Function 778
RANGE Function 779
RANK Function 780
RANNOR Function 780
RANPOI Function 781
RANTBL Function 782
RANTRI Function 783
RANUNI Function 784
REPEAT Function 785
RESOLVE Function 786
REVERSE Function 787
REWIND Function 787
RIGHT Function 788
RMS Function 789
ROUND Function 790
ROUNDE Function 797
ROUNDZ Function 798
RXMATCH Function 801
RXPARSE Function 802
SAVING Function 815
SCAN Function 816
SCANQ Function 817
SDF Function 819
SECOND Function 820
SIGN Function 821
SIN Function 822
SINH Function 823
SKEWNESS Function 823
SLEEP Function 824
SMALLEST Function 825
SOUNDEX Function 827
SPEDIS Function 828
SQRT Function 830
STD Function 831
STDErr Function 831
STFIPS Function 832
STNAME Function 833
STNAMEl Function 834
STRIP Function 835
SUBPAD Function 837
SUBSTR (left of =) Function 838
SUBSTR (right of =) Function 839
SUBSTRN Function 841
SUM Function 844
SYMEXIST Function 845
SYMGET Function 845
SYMGLOBAL Function 846
SYMLocal Function 847
SYSGET Function 847
SYSMSG Function 848
SYSPARM Function 849
SYSPROCESSID Function 850
SYSPROCESSNAME Function 851
SYSPROD Function 852
SYSRC Function 853
SYSTEM Function 853
TAN Function 854
TANH Function 855
TIME Function 856
TIMEPART Function 856
TINV Function 857
TNONCT Function 858
TODAY Function 859
TRANSLATE Function 859
TRANSTAB Function 861
TRANWRD Function 861
TRIGAMMA Function 863
TRIM Function 863
TRIMN Function 865
TRUNC Function 866
UNIFORM Function 867
UPCASE Function 867
URLDECODE Function 868
URLENCODE Function 869
USS Function 870
UUIDGEN Function 871
VAR Function 872
VARFMT Function 872
VARINFMT Function 874
VARMABEL Function 875
VARLEN Function 876
VARNAME Function 877
VARNUM Function 878
VARRAY Function 879
VARRAYX Function 881
VARNTRANSCODE Function 882
VARTYPE Function 882
VERIFY Function 883
VFOMAT Function 884
VFOMATD Function 885
VFOMATDX Function 886
VFOMATN Function 887
VFOMATNX Function 888
VFOMATW Function 890
Definitions of Functions and CALL Routines

Definition of Functions

A SAS function performs a computation or system manipulation on arguments and returns a value. Most functions use arguments supplied by the user, but a few obtain their arguments from the operating environment.

In Base SAS software, you can use SAS functions in DATA step programming statements, in a WHERE expression, in macro language statements, in PROC REPORT, and in Structured Query Language (SQL).

Some statistical procedures also use SAS functions. In addition, some other SAS software products offer functions that you can use in the DATA step. Refer to the documentation that pertains to the specific SAS software product for additional information about these functions.
Definition of CALL Routines

A CALL routine alters variable values or performs other system functions. CALL routines are similar to functions, but differ from functions in that you cannot use them in assignment statements.

All SAS CALL routines are invoked with CALL statements; that is, the name of the routine must appear after the keyword CALL on the CALL statement.

Syntax

Syntax of Functions

The syntax of a function is

\[ function-name \ (argument-1,< \ldots argument-n>) \]
\[ function-name \ (OF \ variable-list) \]
\[ function-name \ (OF \ array-name[^]) \]

where

- \( function-name \) names the function.
- \( argument \) can be a variable name, constant, or any SAS expression, including another function. The number and kind of arguments that SAS allows are described with individual functions. Multiple arguments are separated by a comma.

**Tip:** If the value of an argument is invalid (for example, missing or outside the prescribed range), SAS writes a note to the log indicating that the argument is invalid, sets \_ERROR\_ to 1, and sets the result to a missing value.

**Examples:**

- \( x = \text{max(cash,credit)}; \)
- \( x = \text{sqrt(1500)}; \)
- \( \text{NewCity}=\text{left(upcase(City))}; \)
- \( x = \text{min(YearTemperature-July,YearTemperature-Dec)}; \)
- \( s = \text{repeat('-----+',16)}; \)
- \( x = \text{min((enroll-drop),(enroll-fail))}; \)
- \( \text{dollars}=\text{int(cash)}; \)
- \( \text{if sum(cash,credit)>1000 then} \)
  \( \text{put ‘Goal reached’;} \)

**variable-list**

can be any form of a SAS variable list, including individual variable names. If more than one variable list appears, separate them with a space or with a comma and another OF.

**Examples:**

- \( a = \text{sum(of x y z)}; \)
The following two examples are equivalent.

- \( \text{a}=\text{sum(of x1-x10 y1-y10 z1-z10)}; \)
  \( \text{a}=\text{sum(of x1-x10, of y1-y10, of z1-z10)}; \)
- \( \text{z}=\text{sum(of y1-y10)}; \)

`array-name[*]`

names a currently defined array. Specifying an array in this way causes SAS to treat the array as a list of the variables instead of processing only one element of the array at a time.

**Examples:**

- \( \text{array y{10} y1-y10}; \)
  \( \text{x}=\text{sum(of y{*})}; \)

---

### Syntax of CALL Routines

The syntax of a CALL routine is

- CALL `routine-name (argument-1<, ...argument-n>);`
- CALL `routine-name (OF variable-list);`
- CALL `routine-name (argument-1 | OF variable-list-1 <, ...argument-n | OF variable-list-n>);`

where

- `routine-name`
  names a SAS CALL routine.
- `argument`
  can be a variable name, a constant, any SAS expression, an external module name, an array reference, or a function. Multiple arguments are separated by a comma. The number and kind of arguments that are allowed are described with individual CALL routines in the dictionary section.

**Examples:**

- `call rxsubstr(rx,string,position);`
- `call set(dsid);`
- `call ranbin(Seed_1,n,p,X1);`
- `call label(abc{j},lab);`

- `variable-list`
  can be any form of a SAS variable list, including variable names. If more than one variable list appears, separate them with a space or with a comma and another OF.

**Examples:**

- `call cats(inventory, of y1-y15, of z1-z15);`
  `call catt(of item17-item23 pack17-pack23);`
Using Functions

Restrictions on Function Arguments

If the value of an argument is invalid, SAS prints an error message and sets the result to a missing value. Here are some common restrictions on function arguments:

- Some functions require that their arguments be restricted within a certain range. For example, the argument of the LOG function must be greater than 0.
- Most functions do not permit missing values as arguments. Exceptions include some of the descriptive statistics functions and financial functions.
- In general, the allowed range of the arguments is platform-dependent, such as with the EXP function.
- For some probability functions, combinations of extreme values can cause convergence problems.

Notes on Descriptive Statistic Functions

SAS provides functions that return descriptive statistics. Except for the MISSING function, the functions correspond to the statistics produced by the MEANS procedure. The computing method for each statistic is discussed in the elementary statistics procedures section of the Base SAS Procedures Guide. SAS calculates descriptive statistics for the nonmissing values of the arguments.

Notes on Financial Functions

SAS provides a group of functions that perform financial calculations. The functions are grouped into the following types:

Table 4.1 Types of Financial Functions

<table>
<thead>
<tr>
<th>Function type</th>
<th>Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cashflow</td>
<td>CONVX, CONVXP</td>
<td>calculates convexity for cashflows</td>
</tr>
<tr>
<td></td>
<td>DUR, DURP</td>
<td>calculates modified duration for cashflows</td>
</tr>
<tr>
<td></td>
<td>PVP, YIELDP</td>
<td>calculates present value and yield-to-maturity for a periodic cashflow</td>
</tr>
<tr>
<td>Parameter calculations</td>
<td>COMPOUND</td>
<td>calculates compound interest parameters</td>
</tr>
<tr>
<td></td>
<td>MORT</td>
<td>calculates amortization parameters</td>
</tr>
<tr>
<td>Internal rate of return</td>
<td>INTRR, IRR</td>
<td>calculates the internal rate of return</td>
</tr>
<tr>
<td>Function type</td>
<td>Functions</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------</td>
</tr>
<tr>
<td>Net present and future value</td>
<td>NETPV, NPV</td>
<td>calculates net present and future values</td>
</tr>
<tr>
<td></td>
<td>SAVING</td>
<td>calculates the future value of periodic saving</td>
</tr>
<tr>
<td>Depreciation</td>
<td>DACCxx</td>
<td>calculates the accumulated depreciation up to the specified period</td>
</tr>
<tr>
<td></td>
<td>DEPxxx</td>
<td>calculates depreciation for a single period</td>
</tr>
</tbody>
</table>

### Special Considerations for Depreciation Functions

The period argument for depreciation functions can be fractional for all of the functions except DEPDBSLL and DACCDBSLL. For fractional arguments, the depreciation is prorated between the two consecutive time periods preceding and following the fractional period.

**CAUTION:**

Verify the depreciation method for fractional periods. You must verify whether this method is appropriate to use with fractional periods because many depreciation schedules, specified as tables, have special rules for fractional periods.

### Using DATA Step Functions within Macro Functions

The macro functions `%SYSFUNC` and `%QSYSFUNC` can call DATA step functions to generate text in the macro facility. `%SYSFUNC` and `%QSYSFUNC` have one difference: `%QSYSFUNC` masks special characters and mnemonics and `%SYSFUNC` does not. For more information on these functions, see `%QSYSFUNC` and `%SYSFUNC` in *SAS Macro Language: Reference*.

 `%SYSFUNC` arguments are a single DATA step function and an optional format, as shown in the following examples:

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

You cannot nest DATA step functions within `%SYSFUNC`. However, you can nest `%SYSFUNC` functions that call DATA step functions. For example:

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

All arguments in DATA step functions within `%SYSFUNC` must be separated by commas. You cannot use argument lists that are preceded by the word OF.

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

```sas
dsid=open("sasuser.houses","i");
dsid=open("&mydata","&mode");
%let dsid=%sysfunc(open(sasuser.houses,i));
%let dsid=%sysfunc(open(&mydata,&mode));
```
You can use these functions to call all of the DATA step SAS functions except those that pertain to DATA step variables or processing. These prohibited functions are: DIF, DIM, HBOUND, INPUT, IORCMGS, LAG, LBOUND, MISSING, PUT, RESOLVE, SYMGET, and all of the variable information functions (for example, VLABEL).

Using Functions to Manipulate Files

SAS manipulates files in different ways, depending on whether you use functions or statements. If you use functions such as FOPEN, FGET, and FCLOSE, you have more opportunity to examine and manipulate your data than when you use statements such as INFILE, INPUT, and PUT.

When you use external files, the FOPEN function allocates a buffer called the File Data Buffer (FDB) and opens the external file for reading or updating. The FREAD function reads a record from the external file and copies the data into the FDB. The FGET function then moves the data to the DATA step variables. The function returns a value that you can check with statements or other functions in the DATA step to determine how to further process your data. After the records are processed, the FWRITE function writes the contents of the FDB to the external file, and the FCLOSE function closes the file.

When you use SAS data sets, the OPEN function opens the data set. The FETCH and FETCHOBS functions read observations from an open SAS data set into the Data Set Data Vector (DDV). The GETVARC and GETVARN functions then move the data to DATA step variables. The functions return a value that you can check with statements or other functions in the DATA step to determine how you want to further process your data. After the data is processed, the CLOSE function closes the data set.

For a complete listing of functions and CALL routines, see “Functions and CALL Routines by Category” on page 270. For complete descriptions and examples, see the dictionary section of this book.

Using Random-Number Functions and CALL Routines

Seed Values

Random-number functions and CALL routines generate streams of random numbers from an initial starting point, called a seed, that either the user or the computer clock supplies. A seed must be a nonnegative integer with a value less than $2^{31} - 1$ (or 2,147,483,647). If you use a positive seed, you can always replicate the stream of random numbers by using the same DATA step. If you use zero as the seed, the computer clock initializes the stream, and the stream of random numbers is not replicable.

Each random-number function and CALL routine generates pseudo-random numbers from a specific statistical distribution. Every random-number function requires a seed value expressed as an integer constant or a variable that contains the integer constant. Every CALL routine calls a variable that contains the seed value. Additionally, every CALL routine requires a variable that contains the generated random numbers.

The seed variable must be initialized prior to the first execution of the function or CALL routine. After each execution of a function, the current seed is updated internally, but the value of the seed argument remains unchanged. After each iteration of the CALL routine, however, the seed variable contains the current seed in the stream that generates the next random number. With a function, it is not possible to control the seed values, and, therefore, the random numbers after the initialization.
Comparison of Random-Number Functions and CALL Routines

Except for the NORMAL and UNIFORM functions, which are equivalent to the RANNOR and RANUNI functions, respectively, SAS provides a CALL routine that has the same name as each random-number function. Using CALL routines gives you greater control over the seed values. As an illustration of this control over seed values, all the random-number CALL routines show an example of interrupting the stream of random numbers to change the seed value.

With a CALL routine, you can generate multiple streams of random numbers within a single DATA step. If you supply a different seed value to initialize each of the seed variables, the streams of the generated random numbers are computationally independent. With a function, however, you cannot generate more than one stream by supplying multiple seeds within a DATA step. The following two examples illustrate the difference.

Examples

Example 1: Generating Multiple Streams from a CALL Routine

This example uses the CALL RANUNI routine to generate three streams of random numbers from the uniform distribution with ten numbers each. See the results in Output 4.1.

```sas
options nodate pageno=1 linesize=80 pagesize=60;

data multiple(drop=i);
  retain Seed_1 1298573062 Seed_2 447801538 Seed_3 631280;
  do i=1 to 10;
    call ranuni (Seed_1,X1);
    call ranuni (Seed_2,X2);
    call ranuni (Seed_3,X3);
    output;
  end;
run;

proc print data=multiple;
  title 'Multiple Streams from a CALL Routine';
run;
```
### Example 2: Assigning Values from a Single Stream to Multiple Variables

Using the same three seeds that were used in Example 1, this example uses a function to create three variables. The results that are produced are different from those in Example 1 because the values of all three variables are generated by the first seed. When you use an individual function more than once in a DATA step, the function accepts only the first seed value that you supply and ignores the rest.

```sas
options nodate pageno=1 linesize=80 pagesize=60;

data single(drop=i);
  do i=1 to 3;
    Y1=ranuni(1298573062);
    Y2=ranuni(447801538);
    Y3=ranuni(631280);
    output;
  end;
run;

proc print data=single;
  title 'A Single Stream across Multiple Variables';
run;
```

Output 4.2 shows the results. The values of Y1, Y2, and Y3 in this example come from the same random-number stream generated from the first seed. You can see this by comparing the values by observation across these three variables with the values of X1 in Output 4.1.

### Output 4.2

<table>
<thead>
<tr>
<th>Obs</th>
<th>Y1</th>
<th>Y2</th>
<th>Y3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.64924</td>
<td>0.89471</td>
<td>0.42047</td>
</tr>
<tr>
<td>2</td>
<td>0.20522</td>
<td>0.48638</td>
<td>0.99475</td>
</tr>
<tr>
<td>3</td>
<td>0.47889</td>
<td>0.54169</td>
<td>0.08225</td>
</tr>
</tbody>
</table>
Pattern Matching Using SAS Regular Expressions (RX) and Perl Regular Expressions (PRX)

Definition of Pattern Matching and Regular Expressions

Pattern matching enables you to search for and extract multiple matching patterns from a character string in one step, as well as to make several substitutions in a string in one step. The DATA step supports two kinds of pattern-matching functions and CALL routines:
- SAS regular expressions (RX)
- Perl regular expressions (PRX).

Regular expressions are a pattern language which provides fast tools for parsing large amounts of text. Regular expressions are composed of characters and special characters that are called metacharacters. The asterisk (*) and the question mark (?) are two examples of metacharacters. The asterisk (*) matches zero or more characters, and the question mark (?) matches one or zero characters. For example, if you issue the `ls data*.txt` command from a UNIX shell prompt, UNIX displays all the files in the current directory that begin with the letters “data” and end with the file extension “txt”.

The asterisk (*) and the question mark (?) are a limited form of regular expressions. Perl regular expressions build on the asterisk and the question mark to make searching more powerful and flexible.

Definition of SAS Regular Expression (RX) Functions and CALL Routines

SAS Regular expression (RX) functions and CALL routines refers to a group of functions and CALL routines that uses SAS' regular expression pattern matching to parse character strings. You can search for character strings that have a specific pattern that you specify, and change a matched substring to a different substring. SAS regular expressions are part of the character string matching category for functions and CALL routines. For a short description of these functions and CALL routines, see the “Functions and CALL Routines by Category” on page 270.

Definition of Perl Regular Expression (PRX) Functions and CALL Routines

Perl regular expression (PRX) functions and CALL routines refers to a group of functions and CALL routines that uses a modified version of Perl as a pattern matching language to parse character strings. You can
- search for a pattern of characters within a string
- extract a substring from a string
- search and replace text with other text
- parse large amounts of text, such as Web logs or other text data, more quickly than with SAS regular expressions.

Perl regular expressions are part of the character string matching category for functions and CALL routines. For a short description of these functions and CALL routines, see the “Functions and CALL Routines by Category” on page 270.
Benefits of Using Perl Regular Expressions in the DATA Step

Using Perl regular expressions in the DATA step enhances search and replace options in text. You can use Perl regular expressions to

- validate data
- replace text
- extract a substring from a string
- write Perl debug output to the SAS log.

You can write SAS programs that do not use regular expressions to produce the same results as you do when you use Perl regular expressions. The code without the regular expressions, however, requires more function calls to handle character positions in a string and to manipulate parts of the string.

Perl regular expressions combine most, if not all, of these steps into one expression. The resulting code is

- less prone to error
- easier to maintain
- clearer to read
- more efficient in terms of improving system performance.

Using Perl Regular Expressions in the DATA Step

License Agreement

The following paragraph complies with sections 3 (a) and 4 (c) of the artistic license:

The PRX functions use a modified version of Perl 5.6.1 to perform regular expression compilation and matching. Perl is compiled into a library for use with SAS. The modified and original Perl 5.6.1 files are freely available from the SAS Web site at http://support.sas.com/rnd/base. Each of the modified files has a comment block at the top of the file describing how and when the file was changed. The executables were given non-standard Perl names. The standard version of Perl can be obtained from http://www.perl.com.

Only Perl regular expressions are accessible from the PRX functions. Other parts of the Perl language are not accessible. The modified version of Perl regular expressions does not support

- Perl variables.
- the regular expression options /c, /g, and /o and the /e option with substitutions.
- named characters, which use the \N{name} syntax.
- the metacharacters \pP, \PP, and \X.
- executing Perl code within a regular expression. This includes the syntax (?{code}), (??{code}), and (?p{code}).
- unicode pattern matching.
- using ?PATTERN?. ? is treated like an ordinary regular expression start and end delimiter.
- the metacharacter \G.
- Perl comments between a pattern and replacement text. For example: s{regexp} # perl comment {replacement}.
- matching backslashes with m/\\/. Instead m/\v should be used to match a backslash.
Syntax of Perl Regular Expressions

Perl regular expressions are composed of characters and special characters that are called metacharacters. When performing a match, SAS searches a source string for a substring that matches the Perl regular expression that you specify. Using metacharacters enables SAS to perform special actions when searching for a match:

- If you use the metacharacter \d, SAS matches a digit between 0–9.
- If you use /\dt/, SAS finds the digits in the string “Raleigh, NC 27506”.
- If you use /world/, SAS finds the substring “world” in the string “Hello world!”.

The following table contains a short list of Perl regular expression metacharacters that you can use when you build your code. You can find a complete list of metacharacters at http://www.perldoc.com/perl5.6.1/pod/perlre.html.

<table>
<thead>
<tr>
<th>Metacharacter</th>
<th>Description</th>
</tr>
</thead>
</table>
| \ | marks the next character as either a special character, a literal, a back reference, or an octal escape:  
  - "n" matches the character "n"  
  - "\n" matches a new line character  
  - "\" matches "\"  
  - "\(" matches "(
| | specifies the or condition when you compare alphanumeric strings. |
| ^ | matches the position at the beginning of the input string. |
| $ | matches the position at the end of the input string. |
| * | matches the preceding subexpression zero or more times:  
  - zo* matches "z" and "zoo"  
  - * is equivalent to {0} |
| + | matches the preceding subexpression one or more times:  
  - "zo+" matches "zo" and "zoo"  
  - "zo+" does not match "z"  
  - + is equivalent to {1,} |
| ? | matches the preceding subexpression zero or one time:  
  - "do(es)?" matches the "do" in "do" or "does"  
  - ? is equivalent to [0,1] |
| [n] | n is a non-negative integer that matches exactly n times:  
  - "o[2]" matches the two o’s in "food"  
  - "o[2]" does not match the "o" in "Bob" |
| [n,] | n is a non-negative integer that matches n or more times:  
  - "o[2,]" matches all the o’s in "fooooood"  
  - "o[2,]" does not match the "o" in "Bob"  
  - "o[1,]" is equivalent to "o+"  
  - "o[0,]" is equivalent to "o*" |
### Metacharacter Description

<table>
<thead>
<tr>
<th>Metacharacter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{n,m}</td>
<td>m and n are non-negative integers, where n&lt;=m. They match at least n and at most m times:</td>
</tr>
<tr>
<td></td>
<td>∙ &quot;o{1,3}&quot; matches the first three o’s in &quot;foooooood&quot;</td>
</tr>
<tr>
<td></td>
<td>∙ &quot;o{0,1}&quot; is equivalent to &quot;o?&quot;</td>
</tr>
</tbody>
</table>

*Note:* You cannot put a space between the comma and the numbers.

- period (.) matches any single character except newline. To match any character including newline, use a pattern such as ".\n".
- (pattern) matches a pattern and creates a capture buffer for the match. To retrieve the position and length of the match that is captured, use CALL PRXPOSN. To retrieve the value of the capture buffer, use the PRXPOSN function. To match parentheses characters, use "\(" or "\)".
- x|y matches either x or y:
  - "z|food" matches "z" or "food"
  - "(z|f)ood" matches "zood" or "food"
- [xyz] specifies a character set that matches any one of the enclosed characters:
  - "[abc]" matches the "a" in "plain"
- [^xyz] specifies a negative character set that matches any character that is not enclosed:
  - "[^abc]" matches the "p" in "plain"
- [a-z] specifies a range of characters that matches any character in the range:
  - "[a-z]" matches any lowercase alphabetic character in the range "a" through "z"
- [^a-z] specifies a range of characters that does not match any character in the range:
  - "[^a-z]" matches any character that is not in the range "a" through "z"
- \b matches a word boundary (the position between a word and a space):
  - "er\b" matches the "er" in "never"
  - "er\b" does not match the "er" in "verb"
- \B matches a non-word boundary:
  - "er\B" matches the "er" in "verb"
  - "er\B" does not match the "er" in "never"
- \d matches a digit character that is equivalent to [0-9].
- \D matches a non-digit character that is equivalent to [^0-9].
- \s matches any white space character including space, tab, form feed, and so on, and is equivalent to [\f\n\r\t\v].
Chapter 4

Metacharacter Description

\S matches any character that is not a white space character and is equivalent to [^\f\n\r\v].

\t matches a tab character and is equivalent to "\x09".

\w matches any word character including the underscore and is equivalent to [A-Za-z0-9_].

\W matches any non-word character and is equivalent to[^A-Za-z0-9_].

\num matches num, where num is a positive integer. This is a reference back to captured matches:

□ "(.)\1" matches two consecutive identical characters.

Example 1: Validating Data

You can test for a pattern of characters within a string. For example, you can examine a string to determine whether it contains a correctly formatted telephone number. This type of test is called data validation.

The following example validates a list of phone numbers. To be valid, a phone number must have one of the following forms: (XXX) XXX-XXXX or XXX-XXX-XXXX.

data _null_;  
if _N_ = 1 then do;
   paren = "\((2-9)\d\d\d\) ?(2-9)\d\d-\d\d\d\d";  
dash = "(2-9)\d\d-[2-9]\d\d-\d\d\d\d";  
regexp = "/(" || paren || ")|(" || dash || ")/";  
retain re;
   re = prxparse(regexp);  
if missing(re) then do;
   putlog "ERROR: Invalid regexp " regexp;  
   stop;
end;

length first last home business $ 16;
input first last home business;

if ^prxmatch(re, home) then putlog "NOTE: Invalid home phone number for " first last home;

if ^prxmatch(re, business) then putlog "NOTE: Invalid business phone number for " first last business;

datalines;
Jerome Johnson (919)319-1677 (919)846-2198
Romeo Montague 800-899-2164 360-973-6201
Imani Rashid (508)852-2146 (508)366-9821
Palinor Kent . 919-782-3199
Ruby Archuleta . .
Takei Ito 7042982145 .
The following items correspond to the lines that are numbered in the DATA step that is shown above.

1. Create a DATA step.
2. Build a Perl regular expression to identify a phone number that matches (XXX)XXX-XXXX, and assign the variable PAREN to hold the result. Use the following syntax elements to build the Perl regular expression:
   - \( matches the open parenthesis in the area code.
   - [2–9] matches the digits 2–9. This is the first number in the area code.
   - \d matches a digit. This is the second number in the area code.
   - \d matches a digit. This is the third number in the area code.
   - \) matches the closed parenthesis in the area code.
   - ? matches the space (which is the preceding subexpression) zero or one time. Spaces are significant in Perl regular expressions. They match a space in the text that you are searching. If a space precedes the question mark metacharacter (as it does in this case), the pattern matches either zero spaces or one space in this position in the phone number.
3. Build a Perl regular expression to identify a phone number that matches XXX-XXX-XXXX, and assign the variable DASH to hold the result.
4. Build a Perl regular expression that concatenates the regular expressions for (XXX)XXX-XXXX and XXX—XXX—XXXX. The concatenation enables you to search for both phone number formats from one regular expression.
   - The PAREN and DASH regular expressions are placed within parentheses. The bar metacharacter (|) that is located between PAREN and DASH instructs the compiler to match either pattern. The slashes around the entire pattern tell the compiler where the start and end of the regular expression is located.
5. Pass the Perl regular expression to PRXPARSE and compile the expression. PRXPARSE returns a value to the compiled pattern. Using the value with other Perl regular expression functions and CALL routines enables SAS to perform operations with the compiled Perl regular expression.
6. Use the MISSING function to check whether the regular expression was successfully compiled.
7. Use the PUTLOG statement to write an error message to the SAS log if the regular expression did not compile.
8. Search for a valid home phone number. PRXMATCH uses the value from PRXPARSE along with the search text and returns the position where the regular expression was found in the search text. If there is no match for the home phone number, the PUTLOG statement writes a note to the SAS log.
9. Search for a valid business phone number. PRXMATCH uses the value from PRXPARSE along with the search text and returns the position where the regular expression was found in the search text. If there is no match for the business phone number, the PUTLOG statement writes a note to the SAS log.
The following lines are written to the SAS log:

NOTE: Invalid home phone number for Palinor Kent
NOTE: Invalid home phone number for Ruby Archuleta
NOTE: Invalid business phone number for Ruby Archuleta
NOTE: Invalid home phone number for Takei Ito 7042982145
NOTE: Invalid business phone number for Takei Ito
NOTE: Invalid home phone number for Tom Joad 209/963/2764
NOTE: Invalid business phone number for Tom Joad 2099-66-8474

Example 2: Replacing Text

You can use Perl regular expressions to find specific characters within a string. You can then remove the characters or replace them with other characters. In this example, the two occurrences of the less-than character (<) are replaced by &lt; and the two occurrences of the greater-than character (>) are replaced by &gt;.

data _null_;  
if _N_ = 1 then  
do;  
rettain lt_re gt_re;  
lt_re = prxparse('s/</&lt;/');  
gt_re = prxparse('s/>/>/');  
if missing(lt_re) or missing(gt_re) then  
do;  
   putlog "ERROR: Invalid regexp.";  
   stop;  
end;  
end;  
input;  
call prxchange(lt_re, -1, _infile_);  
call prxchange(gt_re, -1, _infile_);  
put _infile_;  
datalines4;

The bracketing construct ( ... ) creates capture buffers. To refer to the digit’th buffer use \<digit> within the match. Outside the match use "$" instead of "\". (The \<digit> notation works in certain circumstances outside the match. See the warning below about \1 vs $1 for details.) Referring back to another part of the match is called backreference.

The following items correspond to the numbered lines in the DATA step that is shown above.

1 Create a DATA step.
2 Use metacharacters to create a substitution syntax for a Perl regular expression, and compile the expression. The substitution syntax specifies that a less-than character (<) in the input is replaced by the value &lt; in the output.
3 Use metacharacters to create a substitution syntax for a Perl regular expression, and compile the expression. The substitution syntax specifies that a greater-than character (>) in the input is replaced by the value &gt; in the output.
Use the MISSING function to check whether the Perl regular expression compiled without error.

Use the PUTLOG statement to write an error message to the SAS log if neither of the regular expressions was found.

Call the PRXCHANGE routine. Pass the LT_RE pattern-id, and search for and replace all matching patterns. Put the results in _INFILE_ and write the observation to the SAS log.

Call the PRXCHANGE routine. Pass the GT_RE pattern-id, and search for and replace all matching patterns. Put the results in _INFILE_ and write the observation to the SAS log.

The following lines are written to the SAS log:

```
The bracketing construct ( ... ) creates capture buffers. To refer to the digit'th buffer use \&lt;digit&gt; within the match. Outside the match use "$" instead of "\". (The \&lt;digit&gt; notation works in certain circumstances outside the match. See the warning below about \1 vs $1 for details.) Referring back to another part of the match is called a backreference.
```

**Example 3: Extracting a Substring from a String**

You can use Perl regular expressions to find and easily extract text from a string. In this example, the DATA step creates a subset of North Carolina business phone numbers. The program extracts the area code and checks it against a list of area codes for North Carolina.

```sas
data _null_;  
  if _N_ = 1 then do;
    paren = "\(([2-9]\d\d) ?[2-9]\d\-\d\d\d\);  
    dash = "([2-9]\d\d)-[2-9]\d\-\d\d\d\d\);  
    regexp = "/(\| paren \| )\| (\| dash \| )\)/;  
    retain re;
    re = prxparse(regexp);
    if missing(re) then do;
      putlog "ERROR: Invalid regexp " regexp;
      stop;
    end;
    retain areacode_re;
    areacode_re = prxparse("/828|336|704|910|919|252/" );  
    if missing(areacode_re) then do;
      putlog "ERROR: Invalid area code regexp";
      stop;
    end;
  end;
length first last home business $ 16;
length areacode $ 3;
input first last home business;
```
if !prxmatch(re, home) then
    putlog "NOTE: Invalid home phone number for " first last home;

if prxmatch(re, business) then ❶
do;
    which_format = prxparen(re); ❷
    call prxposn(re, which_format, pos, len);
    areacode = substr(business, pos, len);
    if prxmatch(areacode_re, areacode) then ❸
        put "In North Carolina: " first last business;
    end;
else
    putlog "NOTE: Invalid business phone number for " first last business;

datalines;
Jerome Johnson (919)319-1677 (919)846-2198
Romeo Montague 800-899-2164 360-973-6201
Imani Rashid (508)852-2146 (508)366-9821
Palinor Kent 704-782-4673 704-782-3199
Ruby Archuleta 905-384-2839 905-328-3892
Takei Ito 704-298-2145 704-298-4738
Tom Joad 515-372-4829 515-389-2838
;

The following items correspond to the numbered lines in the DATA step that is shown above.

❶ Create a DATA step.

❷ Build a Perl regular expression to identify a phone number that matches (XXX)XXX-XXXX, and assign the variable PAREN to hold the result. Use the following syntax elements to build the Perl regular expression:

\( matches the open parenthesis in the area code. The open parenthesis marks the start of the submatch.

[2–9] matches the digits 2–9. This is the first number in the area code.

\d matches a digit. This is the second number in the area code.

\d matches a digit. This is the third number in the area code.

\) matches the closed parenthesis in the area code. The closed parenthesis marks the end of the submatch.

? matches the space (which is the preceding subexpression) zero or one time. Spaces are significant in Perl regular expressions. They match a space in the text that you are searching. If a space precedes the question mark metacharacter (as it does in this case), the pattern matches either zero spaces or one space in this position in the phone number.

❸ Build a Perl regular expression to identify a phone number that matches XXX-XXX-XXXX, and assign the variable DASH to hold the result.

❹ Build a Perl regular expression that concatenates the regular expressions for (XXX)XXX-XXXX and XXX—XXX—XXXX. The concatenation enables you to search for both phone number formats from one regular expression.
The PAREN and DASH regular expressions are placed within parentheses. The bar metacharacter (|) that is located between PAREN and DASH instructs the compiler to match either pattern. The slashes around the entire pattern tell the compiler where the start and end of the regular expression is located.

5 Pass the Perl regular expression to PRXPARSE and compile the expression. PRXPARSE returns a value to the compiled pattern. Using the value with other Perl regular expression functions and CALL routines enables SAS to perform operations with the compiled Perl regular expression.

6 Use the MISSING function to check whether the Perl regular expression compiled without error.

7 Use the PUTLOG statement to write an error message to the SAS log if the regular expression did not compile.

8 Compile a Perl regular expression that searches a string for a valid North Carolina area code.

9 Search for a valid business phone number.

10 Use the PRXPAREN function to determine which submatch to use. PRXPAREN returns the last submatch that was matched. If an area code matches the form (XXX), PRXPAREN returns the value 2. If an area code matches the form XXX, PRXPAREN returns the value 4.

11 Call the PRXPOSN routine to retrieve the position and length of the submatch.

12 Use the PRXMATCH function to determine whether the area code is a valid North Carolina area code, and write the observation to the log.

The following lines are written to the SAS log:

In North Carolina: Jerome Johnson (919)846-2198
In North Carolina: Palinor Kent 704-782-3199
In North Carolina: Takei Ito 704-298-4738

Writing Perl Debug Output to the SAS Log

The DATA step provides debugging support with the CALL PRXDEBUG routine. CALL PRXDEBUG enables you to turn on and off Perl debug output messages that are sent to the SAS log.

The following example writes Perl debug output to the SAS log.

data _null_
{
   /* CALL PRXDEBUG(1) turns on Perl debug output. */
call prxdebug(1);
putlog 'PRXPARSE: '; 
re = prxparse('/[bc]d(ef*g)+h[ij]k$/');
putlog 'PRXMATCH: '; 
pos = prxmatch(re, 'abcdefg_gh_');
   /* CALL PRXDEBUG(0) turns off Perl debug output. */
call prxdebug(0);
run;
}

SAS writes the following output to the log.
For a detailed explanation of Perl debug output, see “CALL PRXDEBUG Routine” on page 355.

Base SAS Functions for Web Applications

Four functions that manipulate Web-related content are available in Base SAS software. HTMLENCODE and URLENCODE return encoded strings. HTMLDECODE and URLDECODE return decoded strings. For information about Web-based SAS tools, follow the Communities link on the SAS support page, at support.sas.com.

Functions and CALL Routines by Category

Table 4.2 Categories and Descriptions of Functions and CALL Routines

<table>
<thead>
<tr>
<th>Category</th>
<th>Functions and CALL Routines</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>“DIM Function” on page 498</td>
<td>Returns the number of elements in an array</td>
</tr>
<tr>
<td></td>
<td>“HBOUND Function” on page 576</td>
<td>Returns the upper bound of an array</td>
</tr>
<tr>
<td>Category</td>
<td>Functions and CALL Routines</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>----------------------------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>“LBOUND Function” on page 624</td>
<td>Returns the lower bound of an array</td>
</tr>
<tr>
<td>Bitwise Logical Operations</td>
<td>“BAND Function” on page 328</td>
<td>Returns the bitwise logical AND of two arguments</td>
</tr>
<tr>
<td></td>
<td>“BLSHIFT Function” on page 331</td>
<td>Returns the bitwise logical left shift of two arguments</td>
</tr>
<tr>
<td></td>
<td>“BNOT Function” on page 331</td>
<td>Returns the bitwise logical NOT of an argument</td>
</tr>
<tr>
<td></td>
<td>“BOR Function” on page 332</td>
<td>Returns the bitwise logical OR of two arguments</td>
</tr>
<tr>
<td></td>
<td>“BRSHIFT Function” on page 333</td>
<td>Returns the bitwise logical right shift of two arguments</td>
</tr>
<tr>
<td></td>
<td>“BXOR Function” on page 333</td>
<td>Returns the bitwise logical EXCLUSIVE OR of two arguments</td>
</tr>
<tr>
<td>Character String Matching</td>
<td>“CALL PRXCHANGE Routine” on page 353</td>
<td>Performs a pattern-matching replacement</td>
</tr>
<tr>
<td></td>
<td>“CALL PRXDEBUG Routine” on page 355</td>
<td>Enables Perl regular expressions in a DATA step to send debug output to the SAS log</td>
</tr>
<tr>
<td></td>
<td>“CALL PRXFREE Routine” on page 358</td>
<td>Frees unneeded memory that was allocated for a Perl regular expression</td>
</tr>
<tr>
<td></td>
<td>“CALL PRXNEXT Routine” on page 359</td>
<td>Returns the position and length of a substring that matches a pattern and iterates over multiple matches within one string</td>
</tr>
<tr>
<td></td>
<td>“CALL PRXPOSN Routine” on page 361</td>
<td>Returns the start position and length for a capture buffer</td>
</tr>
<tr>
<td></td>
<td>“CALL PRXSUBSTR Routine” on page 363</td>
<td>Returns the position and length of a substring that matches a pattern</td>
</tr>
<tr>
<td></td>
<td>“CALL RXCHANGE Routine” on page 385</td>
<td>Changes one or more substrings that match a pattern</td>
</tr>
<tr>
<td></td>
<td>“CALL RXFREE Routine” on page 386</td>
<td>Frees memory allocated by other regular expression (RX) functions and CALL routines</td>
</tr>
<tr>
<td></td>
<td>“CALL RXSUBSTR Routine” on page 387</td>
<td>Finds the position, length, and score of a substring that matches a pattern</td>
</tr>
<tr>
<td></td>
<td>“PRXCHANGE Function” on page 739</td>
<td>Performs a pattern-matching replacement</td>
</tr>
<tr>
<td></td>
<td>“PRXMATCH Function” on page 743</td>
<td>Searches for a pattern match and returns the position at which the pattern is found</td>
</tr>
<tr>
<td></td>
<td>“PRXPAREN Function” on page 747</td>
<td>Returns the last bracket match for which there is a match in a pattern</td>
</tr>
<tr>
<td></td>
<td>“PRXPARSE Function” on page 749</td>
<td>Compiles a Perl regular expression (PRX) that can be used for pattern matching of a character value</td>
</tr>
<tr>
<td></td>
<td>“PRXPOSN Function” on page 751</td>
<td>Returns the value for a capture buffer</td>
</tr>
<tr>
<td>Category</td>
<td>Functions and CALL Routines</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------------------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Character</td>
<td>“RXMATCH Function” on page 801</td>
<td>Finds the beginning of a substring that matches a pattern</td>
</tr>
<tr>
<td></td>
<td>“RXPARSE Function” on page 802</td>
<td>Parses a pattern</td>
</tr>
<tr>
<td></td>
<td>“ANYALNUM Function” on page 298</td>
<td>Searches a character string for an alphanumeric character and returns the first position at which it is found</td>
</tr>
<tr>
<td></td>
<td>“ANYALPHA Function” on page 300</td>
<td>Searches a character string for an alphabetic character and returns the first position at which it is found</td>
</tr>
<tr>
<td></td>
<td>“ANYCNTRL Function” on page 301</td>
<td>Searches a character string for a control character and returns the first position at which it is found</td>
</tr>
<tr>
<td></td>
<td>“ANYDIGIT Function” on page 302</td>
<td>Searches a character string for a digit and returns the first position at which it is found</td>
</tr>
<tr>
<td></td>
<td>“ANYFIRST Function” on page 304</td>
<td>Searches a character string for a character that is valid as the first character in a SAS variable name under VALIDVARNAME=V7, and returns the first position at which that character is found</td>
</tr>
<tr>
<td></td>
<td>“ANYGRAPH Function” on page 306</td>
<td>Searches a character string for a graphical character and returns the first position at which it is found</td>
</tr>
<tr>
<td></td>
<td>“ANYLOWER Function” on page 307</td>
<td>Searches a character string for a lowercase letter and returns the first position at which it is found</td>
</tr>
<tr>
<td></td>
<td>“ANYNAME Function” on page 309</td>
<td>Searches a character string for a character that is valid in a SAS variable name under VALIDVARNAME=V7, and returns the first position at which that character is found</td>
</tr>
<tr>
<td></td>
<td>“ANYPRINT Function” on page 311</td>
<td>Searches a character string for a printable character and returns the first position at which it is found</td>
</tr>
<tr>
<td></td>
<td>“ANYPUNCT Function” on page 312</td>
<td>Searches a character string for a punctuation character and returns the first position at which it is found</td>
</tr>
<tr>
<td></td>
<td>“ANYSPACE Function” on page 314</td>
<td>Searches a character string for a white-space character (blank, horizontal and vertical tab, carriage return, line feed, form feed) and returns the first position at which it is found</td>
</tr>
<tr>
<td></td>
<td>“ANYUPPER Function” on page 316</td>
<td>Searches a character string for an uppercase letter and returns the first position at which it is found</td>
</tr>
<tr>
<td></td>
<td>“ANYXDIGIT Function” on page 317</td>
<td>Searches a character string for a hexadecimal character that represents a digit and returns the first position at which that character is found</td>
</tr>
<tr>
<td></td>
<td>“BYTE Function” on page 334</td>
<td>Returns one character in the ASCII or the EBCDIC collating sequence</td>
</tr>
<tr>
<td></td>
<td>“CALL CATS Routine” on page 337</td>
<td>Concatenates character strings and removes leading and trailing blanks</td>
</tr>
<tr>
<td></td>
<td>“CALL CATT Routine” on page 338</td>
<td>Concatenates character strings and removes trailing blanks</td>
</tr>
<tr>
<td>Category</td>
<td>Functions and CALL Routines</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td>“CALL CATX Routine” on page 340</td>
<td>Concatenates character strings, removes leading and trailing blanks, and inserts separators</td>
</tr>
<tr>
<td></td>
<td>“CALL COMPCOST Routine” on page 342</td>
<td>Sets the costs of operations for later use by the COMPGED function</td>
</tr>
<tr>
<td></td>
<td>“CALL MISSING Routine” on page 347</td>
<td>Assigns a missing value to the specified character or numeric variables.</td>
</tr>
<tr>
<td></td>
<td>“CALL SCAN Routine” on page 388</td>
<td>Returns the position and length of a given word from a character expression</td>
</tr>
<tr>
<td></td>
<td>“CALL SCANQ Routine” on page 390</td>
<td>Returns the position and length of a given word from a character expression, and ignores delimiters that are enclosed in quotation marks</td>
</tr>
<tr>
<td></td>
<td>“CAT Function” on page 408</td>
<td>Concatenates character strings without removing leading or trailing blanks</td>
</tr>
<tr>
<td></td>
<td>“CATS Function” on page 410</td>
<td>Concatenates character strings and removes leading and trailing blanks</td>
</tr>
<tr>
<td></td>
<td>“CATT Function” on page 412</td>
<td>Concatenates character strings and removes trailing blanks</td>
</tr>
<tr>
<td></td>
<td>“CATX Function” on page 414</td>
<td>Concatenates character strings, removes leading and trailing blanks, and inserts separators</td>
</tr>
<tr>
<td></td>
<td>“CHOOSEC Function” on page 432</td>
<td>Returns a character value that represents the results of choosing from a list of arguments</td>
</tr>
<tr>
<td></td>
<td>“CHOOSEN Function” on page 433</td>
<td>Returns a numeric value that represents the results of choosing from a list of arguments</td>
</tr>
<tr>
<td></td>
<td>“COALESCEC Function” on page 439</td>
<td>Returns the first non-missing value from a list of character arguments.</td>
</tr>
<tr>
<td></td>
<td>“COLLATE Function” on page 440</td>
<td>Returns an ASCII or EBCDIC collating sequence character string</td>
</tr>
<tr>
<td></td>
<td>“COMPARE Function” on page 443</td>
<td>Returns the position of the leftmost character by which two strings differ, or returns 0 if there is no difference</td>
</tr>
<tr>
<td></td>
<td>“COMPBL Function” on page 446</td>
<td>Removes multiple blanks from a character string</td>
</tr>
<tr>
<td></td>
<td>“COMPGED Function” on page 447</td>
<td>Compares two strings by computing the generalized edit distance</td>
</tr>
<tr>
<td></td>
<td>“COMPLEV Function” on page 452</td>
<td>Compares two strings by computing the Levenshtein edit distance</td>
</tr>
<tr>
<td></td>
<td>“COMPRESS Function” on page 456</td>
<td>Removes specific characters from a character string</td>
</tr>
<tr>
<td></td>
<td>“COUNT Function” on page 466</td>
<td>Counts the number of times that a specific substring of characters appears within a character string that you specify</td>
</tr>
<tr>
<td></td>
<td>“COUNTC Function” on page 468</td>
<td>Counts the number of specific characters that either appear or do not appear within a character string that you specify</td>
</tr>
<tr>
<td>Category</td>
<td>Functions and CALL Routines</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td>“DEQUOTE Function” on page 489</td>
<td>Removes matching quotation marks from a character string that begins with an individual quotation mark and deletes everything that is to the right of the closing quotation mark</td>
</tr>
<tr>
<td></td>
<td>“FIND Function” on page 532</td>
<td>Searches for a specific substring of characters within a character string that you specify</td>
</tr>
<tr>
<td></td>
<td>“FINDC Function” on page 534</td>
<td>Searches for specific characters that either appear or do not appear within a character string that you specify</td>
</tr>
<tr>
<td></td>
<td>“IFC Function” on page 583</td>
<td>Returns a character value of an expression based on whether the expression is true, false, or missing</td>
</tr>
<tr>
<td></td>
<td>“IFN Function” on page 585</td>
<td>Returns a numeric value of an expression based on whether the expression is true, false, or missing</td>
</tr>
<tr>
<td></td>
<td>“INDEX Function” on page 588</td>
<td>Searches a character expression for a string of characters</td>
</tr>
<tr>
<td></td>
<td>“INDEXC Function” on page 589</td>
<td>Searches a character expression for specific characters</td>
</tr>
<tr>
<td></td>
<td>“INDEXW Function” on page 590</td>
<td>Searches a character expression for a specified string as a word</td>
</tr>
<tr>
<td></td>
<td>“LEFT Function” on page 625</td>
<td>Left aligns a SAS character expression</td>
</tr>
<tr>
<td></td>
<td>“LENGTH Function” on page 626</td>
<td>Returns the length of a non-blank character string, excluding trailing blanks, and returns 1 for a blank character string</td>
</tr>
<tr>
<td></td>
<td>“LENGTHC Function” on page 627</td>
<td>Returns the length of a character string, including trailing blanks</td>
</tr>
<tr>
<td></td>
<td>“LENGTHM Function” on page 628</td>
<td>Returns the amount of memory (in bytes) that is allocated for a character string</td>
</tr>
<tr>
<td></td>
<td>“LENGTHN Function” on page 630</td>
<td>Returns the length of a non-blank character string, excluding trailing blanks, and returns 0 for a blank character string</td>
</tr>
<tr>
<td></td>
<td>“LOWCASE Function” on page 640</td>
<td>Converts all letters in an argument to lowercase</td>
</tr>
<tr>
<td></td>
<td>“MISSING Function” on page 647</td>
<td>Returns a numeric result that indicates whether the argument contains a missing value</td>
</tr>
<tr>
<td></td>
<td>“NLCINTERAL Function” on page 662</td>
<td>Converts a character string that you specify to a SAS name literal (n-literal)</td>
</tr>
<tr>
<td></td>
<td>“NOTALNUM Function” on page 665</td>
<td>Searches a character string for a non-alphanumeric character and returns the first position at which it is found</td>
</tr>
<tr>
<td></td>
<td>“NOTALPHA Function” on page 666</td>
<td>Searches a character string for a non-alphabetic character and returns the first position at which it is found</td>
</tr>
<tr>
<td>Category</td>
<td>Functions and CALL Routines</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>”NOTCNTRL Function” on page 668</td>
<td>Searches a character string for a character that is not a control character and returns the first position at which it is found</td>
<td></td>
</tr>
<tr>
<td>”NOTDIGIT Function” on page 669</td>
<td>Searches a character string for any character that is not a digit and returns the first position at which that character is found</td>
<td></td>
</tr>
<tr>
<td>”NOTFIRST Function” on page 672</td>
<td>Searches a character string for an invalid first character in a SAS variable name under VALIDVARNAME=V7, and returns the first position at which that character is found</td>
<td></td>
</tr>
<tr>
<td>”NOTGRAPH Function” on page 674</td>
<td>Searches a character string for a non-graphical character and returns the first position at which it is found</td>
<td></td>
</tr>
<tr>
<td>”NOTLOWER Function” on page 676</td>
<td>Searches a character string for a character that is not a lowercase letter and returns the first position at which that character is found</td>
<td></td>
</tr>
<tr>
<td>”NOTNAME Function” on page 677</td>
<td>Searches a character string for an invalid character in a SAS variable name under VALIDVARNAME=V7, and returns the first position at which that character is found</td>
<td></td>
</tr>
<tr>
<td>”NOTPRINT Function” on page 679</td>
<td>Searches a character string for a non-printable character and returns the first position at which it is found</td>
<td></td>
</tr>
<tr>
<td>”NOTPUNCT Function” on page 680</td>
<td>Searches a character string for a character that is not a punctuation character and returns the first position at which it is found</td>
<td></td>
</tr>
<tr>
<td>”NOTSPACE Function” on page 682</td>
<td>Searches a character string for a character that is not a white-space character (blank, horizontal and vertical tab, carriage return, line feed, form feed) and returns the first position at which it is found</td>
<td></td>
</tr>
<tr>
<td>”NOTUPPER Function” on page 684</td>
<td>Searches a character string for a character that is not an uppercase letter and returns the first position at which that character is found</td>
<td></td>
</tr>
<tr>
<td>”NOTXDIGIT Function” on page 685</td>
<td>Searches a character string for a character that is not a hexadecimal digit and returns the first position at which that character is found</td>
<td></td>
</tr>
<tr>
<td>”NVALID Function” on page 688</td>
<td>Checks a character string for validity for use as a SAS variable name in a SAS statement</td>
<td></td>
</tr>
<tr>
<td>”PROPCASE Function” on page 737</td>
<td>Converts all words in an argument to proper case</td>
<td></td>
</tr>
<tr>
<td>”QUOTE Function” on page 763</td>
<td>Adds double quotation marks to a character value</td>
<td></td>
</tr>
<tr>
<td>”RANK Function” on page 780</td>
<td>Returns the position of a character in the ASCII or EBCDIC collating sequence</td>
<td></td>
</tr>
<tr>
<td>”REPEAT Function” on page 785</td>
<td>Repeats a character expression</td>
<td></td>
</tr>
<tr>
<td>”REVERSE Function” on page 787</td>
<td>Reverses a character expression</td>
<td></td>
</tr>
<tr>
<td>Category</td>
<td>Functions and CALL Routines</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------------------------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>“RIGHT Function” on page 788</td>
<td>Right aligns a character expression</td>
</tr>
<tr>
<td></td>
<td>“SCAN Function” on page 816</td>
<td>Selects a given word from a character expression</td>
</tr>
<tr>
<td></td>
<td>“SCANQ Function” on page 817</td>
<td>Returns the nth word from a character expression, ignoring delimiters that are enclosed in quotation marks</td>
</tr>
<tr>
<td></td>
<td>“SOUNDEX Function” on page 827</td>
<td>Encodes a string to facilitate searching</td>
</tr>
<tr>
<td></td>
<td>“SPEDIS Function” on page 828</td>
<td>Determines the likelihood of two words matching, expressed as the asymmetric spelling distance between the two words</td>
</tr>
<tr>
<td></td>
<td>“STRIP Function” on page 835</td>
<td>Returns a character string with all leading and trailing blanks removed</td>
</tr>
<tr>
<td></td>
<td>“SUBPAD Function” on page 837</td>
<td>Returns a substring that has a length you specify, using blank padding if necessary</td>
</tr>
<tr>
<td></td>
<td>“SUBSTR (left of =) Function” on page 838</td>
<td>Replaces character value contents</td>
</tr>
<tr>
<td></td>
<td>“SUBSTR (right of =) Function” on page 839</td>
<td>Extracts a substring from an argument</td>
</tr>
<tr>
<td></td>
<td>“SUBSTRN Function” on page 841</td>
<td>Returns a substring, allowing a result with a length of zero</td>
</tr>
<tr>
<td></td>
<td>“TRANSLATE Function” on page 859</td>
<td>Replaces specific characters in a character expression</td>
</tr>
<tr>
<td></td>
<td>“TRANTAB Function” on page 861</td>
<td>Transcodes a data string by using a translation table</td>
</tr>
<tr>
<td></td>
<td>“TRANWRD Function” on page 861</td>
<td>Replaces or removes all occurrences of a word in a character string</td>
</tr>
<tr>
<td></td>
<td>“TRIM Function” on page 863</td>
<td>Removes trailing blanks from character expressions and returns one blank if the expression is missing</td>
</tr>
<tr>
<td></td>
<td>“TRIMN Function” on page 865</td>
<td>Removes trailing blanks from character expressions and returns a null string (zero blanks) if the expression is missing</td>
</tr>
<tr>
<td></td>
<td>“UPCASE Function” on page 867</td>
<td>Converts all letters in an argument to uppercase</td>
</tr>
<tr>
<td></td>
<td>“VERIFY Function” on page 883</td>
<td>Returns the position of the first character that is unique to an expression</td>
</tr>
<tr>
<td>Currency Conversion</td>
<td>“EUROCURR Function” on page 513</td>
<td>Converts one European currency to another</td>
</tr>
<tr>
<td>DBCS</td>
<td>“KCOMPARE Function” on page 615</td>
<td>Returns the result of a comparison of character strings</td>
</tr>
<tr>
<td></td>
<td>“KCOMPRESS Function” on page 615</td>
<td>Removes specific characters from a character string</td>
</tr>
<tr>
<td>Category</td>
<td>Functions and CALL Routines</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>“KCOUNT Function” on page 616</td>
<td>Returns the number of double-byte characters in a string</td>
</tr>
<tr>
<td></td>
<td>“KCVDT Function” on page 616</td>
<td>Converts data from an encoding code to another encoding code</td>
</tr>
<tr>
<td></td>
<td>“KINDEX Function” on page 616</td>
<td>Searches a character expression for a string of characters</td>
</tr>
<tr>
<td></td>
<td>“KINDEXC Function” on page 616</td>
<td>Searches a character expression for specific characters</td>
</tr>
<tr>
<td></td>
<td>“KLEFT Function” on page 616</td>
<td>Left aligns a character expression by removing unnecessary leading DBCS blanks and SO/SI</td>
</tr>
<tr>
<td></td>
<td>“KLENGTH Function” on page 617</td>
<td>Returns the length of an argument</td>
</tr>
<tr>
<td></td>
<td>“KLOWCASE Function” on page 617</td>
<td>Converts all letters in an argument to lowercase</td>
</tr>
<tr>
<td></td>
<td>“KREVERSE Function” on page 617</td>
<td>Reverses a character expression</td>
</tr>
<tr>
<td></td>
<td>“KRIGHT Function” on page 617</td>
<td>Right aligns a character expression by trimming trailing DBCS blanks and SO/SI</td>
</tr>
<tr>
<td></td>
<td>“KSCAN Function” on page 617</td>
<td>Selects a specific word from a character expression</td>
</tr>
<tr>
<td></td>
<td>“KSTRCAT Function” on page 618</td>
<td>Concatenates two or more character strings</td>
</tr>
<tr>
<td></td>
<td>“KSUBSTR Function” on page 618</td>
<td>Extracts a substring from an argument</td>
</tr>
<tr>
<td></td>
<td>“KSUBSTRB Function” on page 618</td>
<td>Extracts a substring from an argument according to the byte position of the substring in the argument</td>
</tr>
<tr>
<td></td>
<td>“KTRANSLATE Function” on page 618</td>
<td>Replaces specific characters in a character expression</td>
</tr>
<tr>
<td></td>
<td>“KTRIM Function” on page 618</td>
<td>Removes trailing DBCS blanks and SO/SI from character expressions</td>
</tr>
<tr>
<td></td>
<td>“KTRUNCATE Function” on page 619</td>
<td>Truncates a numeric value to a specified length</td>
</tr>
<tr>
<td></td>
<td>“KUPCASE Function” on page 619</td>
<td>Converts all single-byte letters in an argument to uppercase</td>
</tr>
<tr>
<td></td>
<td>“KUPDATE Function” on page 619</td>
<td>Inserts, deletes, and replaces character value contents</td>
</tr>
<tr>
<td></td>
<td>“KUPDATEB Function” on page 619</td>
<td>Inserts, deletes, and replaces the contents of the character value according to the byte position of the character value in the argument</td>
</tr>
<tr>
<td></td>
<td>“KVERIFY Function” on page 620</td>
<td>Returns the position of the first character that is unique to an expression</td>
</tr>
<tr>
<td>Date and Time</td>
<td>“DATDIF Function” on page 477</td>
<td>Returns the number of days between two dates</td>
</tr>
<tr>
<td>Category</td>
<td>Functions and CALL Routines</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td>“DATE Function” on page 478</td>
<td>Returns the current date as a SAS date value</td>
</tr>
<tr>
<td></td>
<td>“DATEJUL Function” on page 479</td>
<td>Converts a Julian date to a SAS date value</td>
</tr>
<tr>
<td></td>
<td>“DATEPART Function” on page 479</td>
<td>Extracts the date from a SAS datetime value</td>
</tr>
<tr>
<td></td>
<td>“DATETIME Function” on page 480</td>
<td>Returns the current date and time of day as a SAS datetime value</td>
</tr>
<tr>
<td></td>
<td>“DAY Function” on page 481</td>
<td>Returns the day of the month from a SAS date value</td>
</tr>
<tr>
<td></td>
<td>“DHMS Function” on page 495</td>
<td>Returns a SAS datetime value from date, hour, minute, and second</td>
</tr>
<tr>
<td></td>
<td>“HMS Function” on page 578</td>
<td>Returns a SAS time value from hour, minute, and second values</td>
</tr>
<tr>
<td></td>
<td>“HOUR Function” on page 579</td>
<td>Returns the hour from a SAS time or datetime value</td>
</tr>
<tr>
<td></td>
<td>“INTCK Function” on page 598</td>
<td>Returns the integer count of the number of interval boundaries between two dates, two times, or two datetime values</td>
</tr>
<tr>
<td></td>
<td>“INTNX Function” on page 602</td>
<td>Increments a date, time, or datetime value by a given interval or intervals, and returns a date, time, or datetime value</td>
</tr>
<tr>
<td></td>
<td>“JULDATE Function” on page 613</td>
<td>Returns the Julian date from a SAS date value</td>
</tr>
<tr>
<td></td>
<td>“JULDATE7 Function” on page 614</td>
<td>Returns a seven-digit Julian date from a SAS date value</td>
</tr>
<tr>
<td></td>
<td>“MDY Function” on page 643</td>
<td>Returns a SAS date value from month, day, and year values</td>
</tr>
<tr>
<td></td>
<td>“MINUTE Function” on page 646</td>
<td>Returns the minute from a SAS time or datetime value</td>
</tr>
<tr>
<td></td>
<td>“MONTH Function” on page 655</td>
<td>Returns the month from a SAS date value</td>
</tr>
<tr>
<td></td>
<td>“NLDATE Function” on page 661</td>
<td>Converts the SAS date value to the date value of the specified locale using the date-format modifiers</td>
</tr>
<tr>
<td></td>
<td>“NLDATM Function” on page 661</td>
<td>Converts the SAS datetime values to the time value of the specified locale using the datetime format modifiers</td>
</tr>
<tr>
<td></td>
<td>“NLTIME Function” on page 661</td>
<td>Converts the SAS time or datetime value to the time value of the specified locale using the time-format modifiers</td>
</tr>
<tr>
<td></td>
<td>“QTR Function” on page 760</td>
<td>Returns the quarter of the year from a SAS date value</td>
</tr>
<tr>
<td></td>
<td>“SECOND Function” on page 820</td>
<td>Returns the second from a SAS time or datetime value</td>
</tr>
<tr>
<td>Category</td>
<td>Functions and CALL Routines</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>“TIME Function” on page 856</td>
<td>Returns the current time of day</td>
</tr>
<tr>
<td></td>
<td>“TIMEPART Function” on page 856</td>
<td>Extracts a time value from a SAS datetime value</td>
</tr>
<tr>
<td></td>
<td>“TODAY Function” on page 859</td>
<td>Returns the current date as a SAS date value</td>
</tr>
<tr>
<td></td>
<td>“WEEK Function” on page 916</td>
<td>Returns the week number value</td>
</tr>
<tr>
<td></td>
<td>“WEEKDAY Function” on page 916</td>
<td>Returns the day of the week from a SAS date value</td>
</tr>
<tr>
<td></td>
<td>“YEAR Function” on page 916</td>
<td>Returns the year from a SAS date value</td>
</tr>
<tr>
<td></td>
<td>“YRDIF Function” on page 918</td>
<td>Returns the difference in years between two dates</td>
</tr>
<tr>
<td></td>
<td>“YYQ Function” on page 920</td>
<td>Returns a SAS date value from the year and quarter</td>
</tr>
<tr>
<td>Descriptive Statistics</td>
<td>“CSS Function” on page 469</td>
<td>Returns the corrected sum of squares</td>
</tr>
<tr>
<td></td>
<td>“CV Function” on page 471</td>
<td>Returns the coefficient of variation</td>
</tr>
<tr>
<td></td>
<td>“GEOMEAN Function” on page 566</td>
<td>Returns the geometric mean</td>
</tr>
<tr>
<td></td>
<td>“GEOMEANZ Function” on page 568</td>
<td>Returns the geometric mean, using zero fuzzing</td>
</tr>
<tr>
<td></td>
<td>“HARMEAN Function” on page 574</td>
<td>Returns the harmonic mean</td>
</tr>
<tr>
<td></td>
<td>“HARMEANZ Function” on page 575</td>
<td>Returns the harmonic mean, using zero fuzzing</td>
</tr>
<tr>
<td></td>
<td>“IQR Function” on page 611</td>
<td>Returns the interquartile range</td>
</tr>
<tr>
<td></td>
<td>“KURTOSIS Function” on page 619</td>
<td>Returns the kurtosis</td>
</tr>
<tr>
<td></td>
<td>“LARGEST Function” on page 623</td>
<td>Returns the kth largest non-missing value</td>
</tr>
<tr>
<td></td>
<td>“MAD Function” on page 641</td>
<td>Returns the median absolute deviation from the median</td>
</tr>
<tr>
<td></td>
<td>“MAX Function” on page 642</td>
<td>Returns the largest value</td>
</tr>
<tr>
<td></td>
<td>“MEAN Function” on page 644</td>
<td>Returns the arithmetic mean (average)</td>
</tr>
<tr>
<td></td>
<td>“MEDIAN Function” on page 645</td>
<td>Computes median values</td>
</tr>
<tr>
<td></td>
<td>“MIN Function” on page 646</td>
<td>Returns the smallest value</td>
</tr>
<tr>
<td>Category</td>
<td>Functions and CALL Routines</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td>“MISSING Function” on page 647</td>
<td>Returns a numeric result that indicates whether the argument contains a missing value</td>
</tr>
<tr>
<td></td>
<td>“N Function” on page 659</td>
<td>Returns the number of nonmissing values</td>
</tr>
<tr>
<td></td>
<td>“NMISS Function” on page 664</td>
<td>Returns the number of missing values</td>
</tr>
<tr>
<td></td>
<td>“ORDINAL Function” on page 692</td>
<td>Returns any specified order statistic</td>
</tr>
<tr>
<td></td>
<td>“PCTL Function” on page 694</td>
<td>Computes percentiles</td>
</tr>
<tr>
<td></td>
<td>“RANGE Function” on page 779</td>
<td>Returns the range of values</td>
</tr>
<tr>
<td></td>
<td>“RMS Function” on page 789</td>
<td>Returns the root mean square</td>
</tr>
<tr>
<td></td>
<td>“SKEWNESS Function” on page 823</td>
<td>Returns the skewness</td>
</tr>
<tr>
<td></td>
<td>“SMALLEST Function” on page 825</td>
<td>Returns the kth smallest nonmissing value</td>
</tr>
<tr>
<td></td>
<td>“STD Function” on page 831</td>
<td>Returns the standard deviation</td>
</tr>
<tr>
<td></td>
<td>“STDERR Function” on page 831</td>
<td>Returns the standard error of the mean</td>
</tr>
<tr>
<td></td>
<td>“SUM Function” on page 844</td>
<td>Returns the sum of the nonmissing arguments</td>
</tr>
<tr>
<td></td>
<td>“USS Function” on page 870</td>
<td>Returns the uncorrected sum of squares</td>
</tr>
<tr>
<td></td>
<td>“VAR Function” on page 872</td>
<td>Returns the variance</td>
</tr>
<tr>
<td>External Files</td>
<td>“DCLOSE Function” on page 481</td>
<td>Closes a directory that was opened by the DOPEN function</td>
</tr>
<tr>
<td></td>
<td>“DINFO Function” on page 499</td>
<td>Returns information about a directory</td>
</tr>
<tr>
<td></td>
<td>“DNUM Function” on page 501</td>
<td>Returns the number of members in a directory</td>
</tr>
<tr>
<td></td>
<td>“DCREATE Function” on page 483</td>
<td>Creates an external directory</td>
</tr>
<tr>
<td></td>
<td>“DOPEN Function” on page 502</td>
<td>Opens a directory and returns a directory identifier value</td>
</tr>
<tr>
<td></td>
<td>“DOPTNAME Function” on page 503</td>
<td>Returns directory attribute information</td>
</tr>
<tr>
<td></td>
<td>“DOPTNUM Function” on page 505</td>
<td>Returns the number of information items that are available for a directory</td>
</tr>
<tr>
<td></td>
<td>“DREAD Function” on page 506</td>
<td>Returns the name of a directory member</td>
</tr>
<tr>
<td>Category</td>
<td>Functions and CALL Routines</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td>“DROPNOTE Function” on page 507</td>
<td>Deletes a note marker from a SAS data set or an external file</td>
</tr>
<tr>
<td></td>
<td>“FAPPEND Function” on page 517</td>
<td>Appends the current record to the end of an external file</td>
</tr>
<tr>
<td></td>
<td>“FCLOSE Function” on page 518</td>
<td>Closes an external file, directory, or directory member</td>
</tr>
<tr>
<td></td>
<td>“FCOL Function” on page 520</td>
<td>Returns the current column position in the File Data Buffer (FDB)</td>
</tr>
<tr>
<td></td>
<td>“FDELETE Function” on page 521</td>
<td>Deletes an external file or an empty directory</td>
</tr>
<tr>
<td></td>
<td>“FEXIST Function” on page 525</td>
<td>Verifies the existence of an external file associated with a fileref</td>
</tr>
<tr>
<td></td>
<td>“FGET Function” on page 526</td>
<td>Copies data from the File Data Buffer (FDB) into a variable</td>
</tr>
<tr>
<td></td>
<td>“FILEEXIST Function” on page 527</td>
<td>Verifies the existence of an external file by its physical name</td>
</tr>
<tr>
<td></td>
<td>“FILENAME Function” on page 528</td>
<td>Assigns or deassigns a fileref to an external file, directory, or output device</td>
</tr>
<tr>
<td></td>
<td>“FILEREF Function” on page 531</td>
<td>Verifies that a fileref has been assigned for the current SAS session</td>
</tr>
<tr>
<td></td>
<td>“FINFO Function” on page 538</td>
<td>Returns the value of a file information item</td>
</tr>
<tr>
<td></td>
<td>“FNOTE Function” on page 547</td>
<td>Identifies the last record that was read and returns a value that FPOINT can use</td>
</tr>
<tr>
<td></td>
<td>“FOPEN Function” on page 549</td>
<td>Opens an external file and returns a file identifier value</td>
</tr>
<tr>
<td></td>
<td>“FOPTNAME Function” on page 551</td>
<td>Returns the name of an item of information about a file</td>
</tr>
<tr>
<td></td>
<td>“FOPTNUM Function” on page 553</td>
<td>Returns the number of information items that are available for an external file</td>
</tr>
<tr>
<td></td>
<td>“FPOINT Function” on page 554</td>
<td>Positions the read pointer on the next record to be read</td>
</tr>
<tr>
<td></td>
<td>“FPOS Function” on page 555</td>
<td>Sets the position of the column pointer in the File Data Buffer (FDB)</td>
</tr>
<tr>
<td></td>
<td>“FPUT Function” on page 556</td>
<td>Moves data to the File Data Buffer (FDB) of an external file, starting at the FDB’s current column position</td>
</tr>
<tr>
<td></td>
<td>“FREAD Function” on page 558</td>
<td>Reads a record from an external file into the File Data Buffer (FDB)</td>
</tr>
<tr>
<td></td>
<td>“FREWIND Function” on page 559</td>
<td>Positions the file pointer to the start of the file</td>
</tr>
<tr>
<td></td>
<td>“FRLEN Function” on page 560</td>
<td>Returns the size of the last record read, or, if the file is opened for output, returns the current record size</td>
</tr>
</tbody>
</table>
## Category Functions and CALL Routines

<table>
<thead>
<tr>
<th>Category</th>
<th>Functions and CALL Routines</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>“FSEP Function” on page 561</td>
<td>Sets the token delimiters for the FGET function</td>
</tr>
<tr>
<td></td>
<td>“FWRITE Function” on page 563</td>
<td>Writes a record to an external file</td>
</tr>
<tr>
<td></td>
<td>“MOPEN Function” on page 656</td>
<td>Opens a file by directory id and member name, and returns the file identifier or a 0</td>
</tr>
<tr>
<td></td>
<td>“PATHNAME Function” on page 692</td>
<td>Returns the physical name of a SAS data library or of an external file, or returns a blank</td>
</tr>
<tr>
<td></td>
<td>“SYMSG Function” on page 848</td>
<td>Returns the text of error messages or warning messages from the last data set or external file function execution</td>
</tr>
<tr>
<td></td>
<td>“SYSRC Function” on page 853</td>
<td>Returns a system error number</td>
</tr>
<tr>
<td>External Routines</td>
<td>“CALL MODULE Routine” on page 348</td>
<td>Calls the external routine without any return code</td>
</tr>
<tr>
<td></td>
<td>“CALL MODULEI Routine” on page 351</td>
<td>Calls the external routine without any return code (in IML environment only)</td>
</tr>
<tr>
<td></td>
<td>“MODULEC Function” on page 651</td>
<td>Calls an external routine and returns a character value</td>
</tr>
<tr>
<td></td>
<td>“MODULEIC Function” on page 651</td>
<td>Calls an external routine and returns a character value (in IML environment only)</td>
</tr>
<tr>
<td></td>
<td>“MODULEIN Function” on page 652</td>
<td>Calls an external routine and returns a numeric value (in IML environment only)</td>
</tr>
<tr>
<td></td>
<td>“MODULEIN Function” on page 652</td>
<td>Calls an external routine and returns a numeric value</td>
</tr>
<tr>
<td>Financial</td>
<td>“COMPOUND Function” on page 455</td>
<td>Returns compound interest parameters</td>
</tr>
<tr>
<td></td>
<td>“CONVX Function” on page 462</td>
<td>Returns the convexity for an enumerated cash flow</td>
</tr>
<tr>
<td></td>
<td>“CONVXP Function” on page 463</td>
<td>Returns the convexity for a periodic cash flow stream, such as a bond</td>
</tr>
<tr>
<td></td>
<td>“DACCDDB Function” on page 472</td>
<td>Returns the accumulated declining balance depreciation</td>
</tr>
<tr>
<td></td>
<td>“DACCDDBSL Function” on page 473</td>
<td>Returns the accumulated declining balance with conversion to a straight-line depreciation</td>
</tr>
<tr>
<td></td>
<td>“DACCSL Function” on page 474</td>
<td>Returns the accumulated straight-line depreciation</td>
</tr>
<tr>
<td></td>
<td>“DACCSDY Function” on page 474</td>
<td>Returns the accumulated sum-of-years-digits depreciation</td>
</tr>
<tr>
<td></td>
<td>“DACCTAB Function” on page 475</td>
<td>Returns the accumulated depreciation from specified tables</td>
</tr>
<tr>
<td></td>
<td>“DEPDB Function” on page 484</td>
<td>Returns the declining balance depreciation</td>
</tr>
<tr>
<td>Category</td>
<td>Functions and CALL Routines</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td>“DEPDBSL Function” on page 485</td>
<td>Returns the declining balance with conversion to a straight-line depreciation</td>
</tr>
<tr>
<td></td>
<td>“DEPSL Function” on page 486</td>
<td>Returns the straight-line depreciation</td>
</tr>
<tr>
<td></td>
<td>“DEPSYD Function” on page 487</td>
<td>Returns the sum-of-years-digits depreciation</td>
</tr>
<tr>
<td></td>
<td>“DEPTAB Function” on page 488</td>
<td>Returns the depreciation from specified tables</td>
</tr>
<tr>
<td></td>
<td>“DUR Function” on page 509</td>
<td>Returns the modified duration for an enumerated cash flow</td>
</tr>
<tr>
<td></td>
<td>“DURP Function” on page 510</td>
<td>Returns the modified duration for a periodic cash flow stream, such as a bond</td>
</tr>
<tr>
<td></td>
<td>“INTRR Function” on page 607</td>
<td>Returns the internal rate of return as a fraction</td>
</tr>
<tr>
<td></td>
<td>“IRR Function” on page 612</td>
<td>Returns the internal rate of return as a percentage</td>
</tr>
<tr>
<td></td>
<td>“MORT Function” on page 658</td>
<td>Returns amortization parameters</td>
</tr>
<tr>
<td></td>
<td>“NETPV Function” on page 660</td>
<td>Returns the net present value as a fraction</td>
</tr>
<tr>
<td></td>
<td>“NPV Function” on page 687</td>
<td>Returns the net present value with the rate expressed as a percentage</td>
</tr>
<tr>
<td></td>
<td>“PVP Function” on page 759</td>
<td>Returns the present value for a periodic cash flow stream with repayment of principal at maturity, such as a bond</td>
</tr>
<tr>
<td></td>
<td>“SAVING Function” on page 815</td>
<td>Returns the future value of a periodic saving</td>
</tr>
<tr>
<td></td>
<td>“YIELDP Function” on page 917</td>
<td>Returns the yield-to-maturity for a periodic cash flow stream, such as a bond</td>
</tr>
<tr>
<td>Hyperbolic</td>
<td>“COSH Function” on page 465</td>
<td>Returns the hyperbolic cosine</td>
</tr>
<tr>
<td></td>
<td>“SINH Function” on page 823</td>
<td>Returns the hyperbolic sine</td>
</tr>
<tr>
<td></td>
<td>“TANH Function” on page 855</td>
<td>Returns the hyperbolic tangent</td>
</tr>
<tr>
<td>Macro</td>
<td>“CALL EXECUTE Routine” on page 344</td>
<td>Resolves an argument and issues the resolved value for execution</td>
</tr>
<tr>
<td></td>
<td>“CALL SYMPUT Routine” on page 401</td>
<td>Assigns DATA step information to a macro variable</td>
</tr>
<tr>
<td></td>
<td>“CALL SYMPUTX Routine” on page 402</td>
<td>Assigns a value to a macro variable and removes both leading and trailing blanks</td>
</tr>
<tr>
<td></td>
<td>“RESOLVE Function” on page 786</td>
<td>Returns the resolved value of an argument after it has been processed by the macro facility</td>
</tr>
<tr>
<td>Category</td>
<td>Functions and CALL Routines</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>“SYMEXIST Function” on page 845</td>
<td>Returns an indication of the existence of a macro variable</td>
</tr>
<tr>
<td></td>
<td>“SYMGET Function” on page 845</td>
<td>Returns the value of a macro variable during DATA step execution</td>
</tr>
<tr>
<td></td>
<td>“SYMGLOBL Function” on page 846</td>
<td>Returns an indication as to whether a macro variable is in a global scope to the DATA step during DATA step execution.</td>
</tr>
<tr>
<td></td>
<td>“SYMLOCAL Function” on page 847</td>
<td>Returns an indication as to whether a macro variable is in a local scope to the DATA step during DATA step execution.</td>
</tr>
<tr>
<td>Mathematical</td>
<td>“ABS Function” on page 294</td>
<td>Returns the absolute value</td>
</tr>
<tr>
<td></td>
<td>“AIRY Function” on page 297</td>
<td>Returns the value of the airy function</td>
</tr>
<tr>
<td></td>
<td>“BETA Function” on page 329</td>
<td>Returns the value of the beta function</td>
</tr>
<tr>
<td></td>
<td>“CALL ALLPERM Routine” on page 335</td>
<td>Generates all permutations of the values of several variables</td>
</tr>
<tr>
<td></td>
<td>“CALL LOGISTIC Routine” on page 346</td>
<td>Returns the logistic value</td>
</tr>
<tr>
<td></td>
<td>“CALL SOFTMAX Routine” on page 395</td>
<td>Returns the softmax value</td>
</tr>
<tr>
<td></td>
<td>“CALL STDIZE Routine” on page 396</td>
<td>Standardizes the values of one or more variables</td>
</tr>
<tr>
<td></td>
<td>“CALL TANH Routine” on page 404</td>
<td>Returns the hyperbolic tangent</td>
</tr>
<tr>
<td></td>
<td>“CNONCT Function” on page 437</td>
<td>Returns the noncentrality parameter from a chi-squared distribution</td>
</tr>
<tr>
<td></td>
<td>“COALESCE Function” on page 438</td>
<td>Returns the first non-missing value from a list of numeric arguments.</td>
</tr>
<tr>
<td></td>
<td>“COMB Function” on page 442</td>
<td>Computes the number of combinations of n elements taken r at a time</td>
</tr>
<tr>
<td></td>
<td>“CONSTANT Function” on page 459</td>
<td>Computes some machine and mathematical constants</td>
</tr>
<tr>
<td></td>
<td>“DAIRY Function” on page 476</td>
<td>Returns the derivative of the AIRY function</td>
</tr>
<tr>
<td></td>
<td>“DEVIANACE Function” on page 491</td>
<td>Computes the deviance</td>
</tr>
<tr>
<td></td>
<td>“DIGAMMA Function” on page 497</td>
<td>Returns the value of the Digamma function</td>
</tr>
<tr>
<td></td>
<td>“ERF Function” on page 511</td>
<td>Returns the value of the (normal) error function</td>
</tr>
<tr>
<td>Category</td>
<td>Functions and CALL Routines</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td>“ERFC Function” on page 512</td>
<td>Returns the value of the complementary (normal) error function</td>
</tr>
<tr>
<td></td>
<td>“EXP Function” on page 515</td>
<td>Returns the value of the exponential function</td>
</tr>
<tr>
<td></td>
<td>“FACT Function” on page 516</td>
<td>Computes a factorial</td>
</tr>
<tr>
<td></td>
<td>“FNONCT Function” on page 546</td>
<td>Returns the value of the noncentrality parameter of an F distribution</td>
</tr>
<tr>
<td></td>
<td>“GAMMA Function” on page 565</td>
<td>Returns the value of the Gamma function</td>
</tr>
<tr>
<td></td>
<td>“IBESSEL Function” on page 583</td>
<td>Returns the value of the modified bessel function</td>
</tr>
<tr>
<td></td>
<td>“JBESSEL Function” on page 613</td>
<td>Returns the value of the bessel function</td>
</tr>
<tr>
<td></td>
<td>“LGAMMA Function” on page 631</td>
<td>Returns the natural logarithm of the Gamma function</td>
</tr>
<tr>
<td></td>
<td>“LOG Function” on page 634</td>
<td>Returns the natural (base e) logarithm</td>
</tr>
<tr>
<td></td>
<td>“LOG10 Function” on page 634</td>
<td>Returns the logarithm to the base 10</td>
</tr>
<tr>
<td></td>
<td>“LOG2 Function” on page 635</td>
<td>Returns the logarithm to the base 2</td>
</tr>
<tr>
<td></td>
<td>“LOGBETA Function” on page 635</td>
<td>Returns the logarithm of the beta function</td>
</tr>
<tr>
<td></td>
<td>“MOD Function” on page 649</td>
<td>Returns the remainder from the division of the first argument by the second argument, fuzzed to avoid most unexpected floating-point results</td>
</tr>
<tr>
<td></td>
<td>“MODZ Function” on page 653</td>
<td>Returns the remainder from the division of the first argument by the second argument, using zero fuzzing</td>
</tr>
<tr>
<td></td>
<td>“PERM Function” on page 714</td>
<td>Computes the number of permutations of n items taken r at a time</td>
</tr>
<tr>
<td></td>
<td>“SIGN Function” on page 821</td>
<td>Returns the sign of a value</td>
</tr>
<tr>
<td></td>
<td>“SQRT Function” on page 830</td>
<td>Returns the square root of a value</td>
</tr>
<tr>
<td></td>
<td>“TNONCT Function” on page 858</td>
<td>Returns the value of the noncentrality parameter from the student’s t distribution</td>
</tr>
<tr>
<td></td>
<td>“TRIGAMMA Function” on page 863</td>
<td>Returns the value of the Trigamma function</td>
</tr>
<tr>
<td>Probability</td>
<td>“CDF Function” on page 416</td>
<td>Computes cumulative distribution functions</td>
</tr>
<tr>
<td>Probability</td>
<td>“LOGCDF Function” on page 636</td>
<td>Computes the logarithm of a left cumulative distribution function</td>
</tr>
<tr>
<td>Category</td>
<td>Functions and CALL Routines</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td>“LOGPDF Function” on page 638</td>
<td>Computes the logarithm of a probability density (mass) function</td>
</tr>
<tr>
<td></td>
<td>“LOGSDF Function” on page 639</td>
<td>Computes the logarithm of a survival function</td>
</tr>
<tr>
<td></td>
<td>“PDF Function” on page 695</td>
<td>Computes probability density (mass) functions</td>
</tr>
<tr>
<td></td>
<td>“POISSON Function” on page 716</td>
<td>Returns the probability from a Poisson distribution</td>
</tr>
<tr>
<td></td>
<td>“PROBBETA Function” on page 717</td>
<td>Returns the probability from a beta distribution</td>
</tr>
<tr>
<td></td>
<td>“PROBBNML Function” on page 718</td>
<td>Returns the probability from a binomial distribution</td>
</tr>
<tr>
<td></td>
<td>“PROBBNRM Function” on page 719</td>
<td>Computes a probability from the bivariate normal distribution</td>
</tr>
<tr>
<td></td>
<td>“PROBCHI Function” on page 720</td>
<td>Returns the probability from a chi-squared distribution</td>
</tr>
<tr>
<td></td>
<td>“PROBF Function” on page 721</td>
<td>Returns the probability from an $F$ distribution</td>
</tr>
<tr>
<td></td>
<td>“PROBGAM Function” on page 722</td>
<td>Returns the probability from a gamma distribution</td>
</tr>
<tr>
<td></td>
<td>“PROBHYPR Function” on page 723</td>
<td>Returns the probability from a hypergeometric distribution</td>
</tr>
<tr>
<td></td>
<td>“PROBMC Function” on page 724</td>
<td>Computes a probability or a quantile from various distributions for multiple comparisons of means</td>
</tr>
<tr>
<td></td>
<td>“PROBNGB Function” on page 735</td>
<td>Returns the probability from a negative binomial distribution</td>
</tr>
<tr>
<td></td>
<td>“PROBNORM Function” on page 736</td>
<td>Returns the probability from the standard normal distribution</td>
</tr>
<tr>
<td></td>
<td>“PROBNT Function” on page 736</td>
<td>Returns the probability from a $t$ distribution</td>
</tr>
<tr>
<td></td>
<td>“SDF Function” on page 819</td>
<td>Computes a survival function</td>
</tr>
<tr>
<td></td>
<td>“BETAINV Function” on page 330</td>
<td>Returns a quantile from the beta distribution</td>
</tr>
<tr>
<td></td>
<td>“CINV Function” on page 434</td>
<td>Returns a quantile from the chi-squared distribution</td>
</tr>
<tr>
<td></td>
<td>“FINV Function” on page 539</td>
<td>Returns a quantile from the $F$ distribution</td>
</tr>
<tr>
<td></td>
<td>“GAMINV Function” on page 565</td>
<td>Returns a quantile from the gamma distribution</td>
</tr>
<tr>
<td></td>
<td>“PROBIT Function” on page 724</td>
<td>Returns a quantile from the standard normal distribution</td>
</tr>
<tr>
<td>Category</td>
<td>Functions and CALL Routines</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>---------------------------------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>“QUANTILE Function” on page 761</td>
<td>Computes the quantile from a specified distribution</td>
</tr>
<tr>
<td></td>
<td>“TINV Function” on page 857</td>
<td>Returns a quantile from the $t$ distribution</td>
</tr>
<tr>
<td>Random Number</td>
<td>“CALL RANBIN Routine” on page 366</td>
<td>Returns a random variate from a binomial distribution</td>
</tr>
<tr>
<td></td>
<td>“CALL RANCAU Routine” on page 368</td>
<td>Returns a random variate from a Cauchy distribution</td>
</tr>
<tr>
<td></td>
<td>“CALL RANEXP Routine” on page 369</td>
<td>Returns a random variate from an exponential distribution</td>
</tr>
<tr>
<td></td>
<td>“CALL RANGAM Routine” on page 371</td>
<td>Returns a random variate from a gamma distribution</td>
</tr>
<tr>
<td></td>
<td>“CALL RANNOR Routine” on page 373</td>
<td>Returns a random variate from a normal distribution</td>
</tr>
<tr>
<td></td>
<td>“CALL RANPERK Routine” on page 375</td>
<td>Randomly permutes the values of the arguments, and returns a permutation of $k$ out of $n$ values</td>
</tr>
<tr>
<td></td>
<td>“CALL RANPERM Routine” on page 376</td>
<td>Randomly permutes the values of the arguments</td>
</tr>
<tr>
<td></td>
<td>“CALL RANPOI Routine” on page 377</td>
<td>Returns a random variate from a Poisson distribution</td>
</tr>
<tr>
<td></td>
<td>“CALL RANTBL Routine” on page 379</td>
<td>Returns a random variate from a tabled probability distribution</td>
</tr>
<tr>
<td></td>
<td>“CALL RANTRI Routine” on page 381</td>
<td>Returns a random variate from a triangular distribution</td>
</tr>
<tr>
<td></td>
<td>“CALL RANUNI Routine” on page 383</td>
<td>Returns a random variate from a uniform distribution</td>
</tr>
<tr>
<td></td>
<td>“CALL STREAMINIT Routine” on page 400</td>
<td>Specifies a seed value to use for subsequent random number generation by the RAND function</td>
</tr>
<tr>
<td></td>
<td>“NORMAL Function” on page 664</td>
<td>Returns a random variate from a normal distribution</td>
</tr>
<tr>
<td></td>
<td>“RANBIN Function” on page 764</td>
<td>Returns a random variate from a binomial distribution</td>
</tr>
<tr>
<td></td>
<td>“RANCAU Function” on page 765</td>
<td>Returns a random variate from a Cauchy distribution</td>
</tr>
<tr>
<td></td>
<td>“RAND Function” on page 766</td>
<td>Generates random numbers from a specified distribution</td>
</tr>
<tr>
<td></td>
<td>“RANEXP Function” on page 777</td>
<td>Returns a random variate from an exponential distribution</td>
</tr>
<tr>
<td></td>
<td>“RANGAM Function” on page 778</td>
<td>Returns a random variate from a gamma distribution</td>
</tr>
<tr>
<td></td>
<td>“RANNOR Function” on page 780</td>
<td>Returns a random variate from a normal distribution</td>
</tr>
<tr>
<td>Category</td>
<td>Functions and CALL Routines</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------------------------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>“RANPOI Function” on page 781</td>
<td>Returns a random variate from a Poisson distribution</td>
</tr>
<tr>
<td></td>
<td>“RANTBL Function” on page 782</td>
<td>Returns a random variate from a tabled probability distribution</td>
</tr>
<tr>
<td></td>
<td>“RANTRI Function” on page 783</td>
<td>Returns a random variate from a triangular distribution</td>
</tr>
<tr>
<td></td>
<td>“RANUNI Function” on page 784</td>
<td>Returns a random variate from a uniform distribution</td>
</tr>
<tr>
<td></td>
<td>“UNIFORM Function” on page 867</td>
<td>Returns a random variate from a uniform distribution</td>
</tr>
<tr>
<td>SAS File I/O</td>
<td>“ATTRC Function” on page 322</td>
<td>Returns the value of a character attribute for a SAS data set</td>
</tr>
<tr>
<td></td>
<td>“ATTRN Function” on page 324</td>
<td>Returns the value of a numeric attribute for the specified SAS data set</td>
</tr>
<tr>
<td></td>
<td>“CEXIST Function” on page 431</td>
<td>Verifies the existence of a SAS catalog or SAS catalog entry</td>
</tr>
<tr>
<td></td>
<td>“CLOSE Function” on page 436</td>
<td>Closes a SAS data set</td>
</tr>
<tr>
<td></td>
<td>“CUROBS Function” on page 470</td>
<td>Returns the observation number of the current observation</td>
</tr>
<tr>
<td></td>
<td>“DROPNOTE Function” on page 507</td>
<td>Deletes a note marker from a SAS data set or an external file</td>
</tr>
<tr>
<td></td>
<td>“DSNAME Function” on page 508</td>
<td>Returns the SAS data set name that is associated with a data set identifier</td>
</tr>
<tr>
<td></td>
<td>“EXIST Function” on page 513</td>
<td>Verifies the existence of a SAS data library member</td>
</tr>
<tr>
<td></td>
<td>“FETCH Function” on page 522</td>
<td>Reads the next nondeleted observation from a SAS data set into the Data Set Data Vector (DDV)</td>
</tr>
<tr>
<td></td>
<td>“FETCHOBS Function” on page 523</td>
<td>Reads a specified observation from a SAS data set into the Data Set Data Vector (DDV)</td>
</tr>
<tr>
<td></td>
<td>“GETVARC Function” on page 571</td>
<td>Returns the value of a SAS data set character variable</td>
</tr>
<tr>
<td></td>
<td>“GETVARN Function” on page 572</td>
<td>Returns the value of a SAS data set numeric variable</td>
</tr>
<tr>
<td></td>
<td>“IORCMSG Function” on page 610</td>
<td>Returns a formatted error message for <em>IORC</em></td>
</tr>
<tr>
<td></td>
<td>“LIBNAME Function” on page 632</td>
<td>Assigns or deassigns a libref for a SAS data library</td>
</tr>
<tr>
<td></td>
<td>“LIBREF Function” on page 633</td>
<td>Verifies that a libref has been assigned</td>
</tr>
<tr>
<td></td>
<td>“NOTE Function” on page 671</td>
<td>Returns an observation ID for the current observation of a SAS data set</td>
</tr>
<tr>
<td>Category</td>
<td>Functions and CALL Routines</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>----------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td>“OPEN Function” on page 690</td>
<td>Opens a SAS data set</td>
</tr>
<tr>
<td></td>
<td>“PATHNAME Function” on page 692</td>
<td>Returns the physical name of a SAS data library or of an external file, or returns a blank</td>
</tr>
<tr>
<td></td>
<td>“POINT Function” on page 715</td>
<td>Locates an observation identified by the NOTE function</td>
</tr>
<tr>
<td></td>
<td>“REWIND Function” on page 787</td>
<td>Positions the data set pointer at the beginning of a SAS data set</td>
</tr>
<tr>
<td></td>
<td>“SYSMSG Function” on page 848</td>
<td>Returns the text of error messages or warning messages from the last data set or external file function execution</td>
</tr>
<tr>
<td></td>
<td>“SYSRC Function” on page 853</td>
<td>Returns a system error number</td>
</tr>
<tr>
<td></td>
<td>“VARFMT Function” on page 872</td>
<td>Returns the format assigned to a SAS data set variable</td>
</tr>
<tr>
<td></td>
<td>“VARINFMT Function” on page 874</td>
<td>Returns the informat assigned to a SAS data set variable</td>
</tr>
<tr>
<td></td>
<td>“VARLABEL Function” on page 875</td>
<td>Returns the label assigned to a SAS data set variable</td>
</tr>
<tr>
<td></td>
<td>“VARLEN Function” on page 876</td>
<td>Returns the length of a SAS data set variable</td>
</tr>
<tr>
<td></td>
<td>“VARNNAME Function” on page 877</td>
<td>Returns the name of a SAS data set variable</td>
</tr>
<tr>
<td></td>
<td>“VARNUM Function” on page 878</td>
<td>Returns the number of a variable’s position in a SAS data set</td>
</tr>
<tr>
<td></td>
<td>“VARTYPE Function” on page 882</td>
<td>Returns the data type of a SAS data set variable</td>
</tr>
<tr>
<td>Special</td>
<td>“ADDR Function” on page 295</td>
<td>Returns the memory address of a numeric variable on a 32–bit platform</td>
</tr>
<tr>
<td></td>
<td>“ADDRLONG Function” on page 296</td>
<td>Returns the memory address of a character variable on 32-bit and 64-bit platforms</td>
</tr>
<tr>
<td></td>
<td>“CALL POKE Routine” on page 352</td>
<td>Writes a value directly into memory on a 32–bit platform</td>
</tr>
<tr>
<td></td>
<td>“CALL POKELONG Routine” on page 353</td>
<td>Writes a value directly into memory on 32-bit and 64-bit platforms</td>
</tr>
<tr>
<td></td>
<td>“CALL SLEEP Routine” on page 394</td>
<td>Suspends the execution of a program that invokes this call routine for a specified period of time</td>
</tr>
<tr>
<td></td>
<td>“CALL SYSTEM Routine” on page 403</td>
<td>Submits an operating environment command for execution</td>
</tr>
<tr>
<td></td>
<td>“DIF Function” on page 496</td>
<td>Returns differences between the argument and its n-th lag</td>
</tr>
<tr>
<td></td>
<td>“GETOPTION Function” on page 569</td>
<td>Returns the value of a SAS system or graphics option</td>
</tr>
<tr>
<td>Category</td>
<td>Functions and CALL Routines</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>---------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>“INPUT Function” on page 592</td>
<td>Returns the value produced when a SAS expression that uses a specified informat expression is read</td>
</tr>
<tr>
<td></td>
<td>“INPUTC Function” on page 594</td>
<td>Enables you to specify a character informat at run time</td>
</tr>
<tr>
<td></td>
<td>“INPUTN Function” on page 596</td>
<td>Enables you to specify a numeric informat at run time</td>
</tr>
<tr>
<td></td>
<td>“LAG Function” on page 620</td>
<td>Returns values from a queue</td>
</tr>
<tr>
<td></td>
<td>“PEEK Function” on page 708</td>
<td>Stores the contents of a memory address into a numeric variable on a 32–bit platform</td>
</tr>
<tr>
<td></td>
<td>“PEEKC Function” on page 709</td>
<td>Stores the contents of a memory address in a character variable on a 32–bit platform</td>
</tr>
<tr>
<td></td>
<td>“PEEKCLONG Function” on page 712</td>
<td>Stores the contents of a memory address in a character variable on 32-bit and 64-bit platforms</td>
</tr>
<tr>
<td></td>
<td>“PEEKLONG Function” on page 713</td>
<td>Stores the contents of a memory address in a numeric variable on 32-bit and 64-bit platforms</td>
</tr>
<tr>
<td></td>
<td>“PTRLONGADD Function” on page 754</td>
<td>Returns the pointer address as a character variable on 32-bit and 64-bit platforms</td>
</tr>
<tr>
<td></td>
<td>“PUT Function” on page 754</td>
<td>Returns a value using a specified format</td>
</tr>
<tr>
<td></td>
<td>“PUTC Function” on page 756</td>
<td>Enables you to specify a character format at run time</td>
</tr>
<tr>
<td></td>
<td>“PUTN Function” on page 758</td>
<td>Enables you to specify a numeric format at run time</td>
</tr>
<tr>
<td></td>
<td>“SLEEP Function” on page 824</td>
<td>Suspends the execution of a program that invokes this function for a specified period of time</td>
</tr>
<tr>
<td></td>
<td>“SYSGET Function” on page 847</td>
<td>Returns the value of the specified operating environment variable</td>
</tr>
<tr>
<td></td>
<td>“SYSPARM Function” on page 849</td>
<td>Returns the system parameter string</td>
</tr>
<tr>
<td></td>
<td>“SYSPROCESSID Function” on page 850</td>
<td>Returns the process id of the current process</td>
</tr>
<tr>
<td></td>
<td>“SYSPROCESSNAME Function” on page 851</td>
<td>Returns the process name associated with a given process id or the name of the current process</td>
</tr>
<tr>
<td></td>
<td>“SYSPROD Function” on page 852</td>
<td>Determines if a product is licensed</td>
</tr>
<tr>
<td></td>
<td>“SYSTEM Function” on page 853</td>
<td>Issues an operating environment command during a SAS session and returns the system return code</td>
</tr>
<tr>
<td></td>
<td>“UUIDGEN Function” on page 871</td>
<td>Returns the short or binary form of a Universal Unique Identifier (UUID)</td>
</tr>
<tr>
<td>State and ZIP Code</td>
<td>“FIPNAME Function” on page 540</td>
<td>Converts two-digit FIPS codes to uppercase state names</td>
</tr>
<tr>
<td>Category</td>
<td>Functions and CALL Routines</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------------------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>“FIPNAMEL Function” on page 541</td>
<td>Converts two-digit FIPS codes to mixed case state names</td>
</tr>
<tr>
<td></td>
<td>“FIPSTATE Function” on page 542</td>
<td>Converts two-digit FIPS codes to two-character state postal codes</td>
</tr>
<tr>
<td></td>
<td>“STFIPS Function” on page 832</td>
<td>Converts state postal codes to FIPS state codes</td>
</tr>
<tr>
<td></td>
<td>“STNAME Function” on page 833</td>
<td>Converts state postal codes to uppercase state names</td>
</tr>
<tr>
<td></td>
<td>“STNAMEL Function” on page 834</td>
<td>Converts state postal codes to mixed case state names</td>
</tr>
<tr>
<td></td>
<td>“ZIPCITY Function” on page 921</td>
<td>Returns a city name and the two-character postal code that corresponds to a ZIP code</td>
</tr>
<tr>
<td></td>
<td>“ZIPFIPS Function” on page 922</td>
<td>Converts ZIP codes to two-digit FIPS codes</td>
</tr>
<tr>
<td></td>
<td>“ZIPNAME Function” on page 924</td>
<td>Converts ZIP codes to uppercase state names</td>
</tr>
<tr>
<td></td>
<td>“ZIPNAMEL Function” on page 925</td>
<td>Converts ZIP codes to mixed case state names</td>
</tr>
<tr>
<td></td>
<td>“ZIPSTATE Function” on page 926</td>
<td>Converts ZIP codes to two-character state postal codes</td>
</tr>
<tr>
<td>Trigonometric</td>
<td>“ARCOS Function” on page 319</td>
<td>Returns the arccosine</td>
</tr>
<tr>
<td></td>
<td>“ARSIN Function” on page 319</td>
<td>Returns the arcsine</td>
</tr>
<tr>
<td></td>
<td>“ATAN Function” on page 320</td>
<td>Returns the arc tangent</td>
</tr>
<tr>
<td></td>
<td>“ATAN2 Function” on page 321</td>
<td>Returns the arc tangent of two numeric variables</td>
</tr>
<tr>
<td></td>
<td>“COS Function” on page 465</td>
<td>Returns the cosine</td>
</tr>
<tr>
<td></td>
<td>“SIN Function” on page 822</td>
<td>Returns the sine</td>
</tr>
<tr>
<td></td>
<td>“TAN Function” on page 854</td>
<td>Returns the tangent</td>
</tr>
<tr>
<td>Truncation</td>
<td>“CEIL Function” on page 429</td>
<td>Returns the smallest integer that is greater than or equal to the argument, fuzzed to avoid unexpected floating-point results</td>
</tr>
<tr>
<td></td>
<td>“CEILZ Function” on page 430</td>
<td>Returns the smallest integer that is greater than or equal to the argument, using zero fuzzing</td>
</tr>
<tr>
<td></td>
<td>“FLOOR Function” on page 544</td>
<td>Returns the largest integer that is less than or equal to the argument, fuzzed to avoid unexpected floating-point results</td>
</tr>
<tr>
<td>Category</td>
<td>Functions and CALL Routines</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>“FLOORZ Function” on page 545</td>
<td>Returns the largest integer that is less than or equal to the argument, using zero fuzzing</td>
</tr>
<tr>
<td></td>
<td>“FUZZ Function” on page 562</td>
<td>Returns the nearest integer if the argument is within 1E–12</td>
</tr>
<tr>
<td></td>
<td>“INT Function” on page 597</td>
<td>Returns the integer value, fuzzed to avoid unexpected floating-point results</td>
</tr>
<tr>
<td></td>
<td>“INTZ Function” on page 608</td>
<td>Returns the integer portion of the argument, using zero fuzzing</td>
</tr>
<tr>
<td></td>
<td>“ROUND Function” on page 790</td>
<td>Rounds the first argument to the nearest multiple of the second argument, or to the nearest integer when the second argument is omitted</td>
</tr>
<tr>
<td></td>
<td>“ROUNDE Function” on page 797</td>
<td>Rounds the first argument to the nearest multiple of the second argument, and returns an even multiple when the first argument is halfway between the two nearest multiples</td>
</tr>
<tr>
<td></td>
<td>“ROUNDEZ Function” on page 798</td>
<td>Rounds the first argument to the nearest multiple of the second argument, with zero fuzzing</td>
</tr>
<tr>
<td></td>
<td>“TRUNC Function” on page 866</td>
<td>Truncates a numeric value to a specified length</td>
</tr>
<tr>
<td>Variable Control</td>
<td>“CALL LABEL Routine” on page 345</td>
<td>Assigns a variable label to a specified character variable</td>
</tr>
<tr>
<td></td>
<td>“CALL SET Routine” on page 392</td>
<td>Links SAS data set variables to DATA step or macro variables that have the same name and data type</td>
</tr>
<tr>
<td></td>
<td>“CALL VNAME Routine” on page 405</td>
<td>Assigns a variable name as the value of a specified variable</td>
</tr>
<tr>
<td>Variable Information</td>
<td>“CALL VNEXT Routine” on page 406</td>
<td>Returns the name, type, and length of a variable that is used in a DATA step</td>
</tr>
<tr>
<td></td>
<td>“VARRAY Function” on page 879</td>
<td>Returns a value that indicates whether the specified name is an array</td>
</tr>
<tr>
<td></td>
<td>“VARRAYX Function” on page 881</td>
<td>Returns a value that indicates whether the value of the specified argument is an array</td>
</tr>
<tr>
<td></td>
<td>“VARTRANSCODE Function” on page 882</td>
<td>Returns the transcode attribute of a SAS data set variable</td>
</tr>
<tr>
<td></td>
<td>“VFORMAT Function” on page 884</td>
<td>Returns the format that is associated with the specified variable</td>
</tr>
<tr>
<td></td>
<td>“VFORMATD Function” on page 885</td>
<td>Returns the format decimal value that is associated with the specified variable</td>
</tr>
<tr>
<td></td>
<td>“VFORMATDX Function” on page 886</td>
<td>Returns the format decimal value that is associated with the value of the specified argument</td>
</tr>
<tr>
<td></td>
<td>“VFORMATN Function” on page 887</td>
<td>Returns the format name that is associated with the specified variable</td>
</tr>
<tr>
<td></td>
<td>“VFORMATNX Function” on page 888</td>
<td>Returns the format name that is associated with the value of the specified argument</td>
</tr>
<tr>
<td>Category</td>
<td>Functions and CALL Routines</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td>“VFORMATW Function” on page 890</td>
<td>Returns the format width that is associated with the specified variable</td>
</tr>
<tr>
<td></td>
<td>“VFORMATWX Function” on page 891</td>
<td>Returns the format width that is associated with the value of the specified argument</td>
</tr>
<tr>
<td></td>
<td>“VFORMATX Function” on page 892</td>
<td>Returns the format that is associated with the value of the specified argument</td>
</tr>
<tr>
<td></td>
<td>“VINARRAY Function” on page 893</td>
<td>Returns a value that indicates whether the specified variable is a member of an array</td>
</tr>
<tr>
<td></td>
<td>“VINARRAYX Function” on page 894</td>
<td>Returns a value that indicates whether the value of the specified argument is a member of an array</td>
</tr>
<tr>
<td></td>
<td>“VINFORMAT Function” on page 895</td>
<td>Returns the informat that is associated with the specified variable</td>
</tr>
<tr>
<td></td>
<td>“VINFORMATD Function” on page 896</td>
<td>Returns the informat decimal value that is associated with the specified variable</td>
</tr>
<tr>
<td></td>
<td>“VINFORMATDX Function” on page 897</td>
<td>Returns the informat decimal value that is associated with the value of the specified argument</td>
</tr>
<tr>
<td></td>
<td>“VINFORMATN Function” on page 898</td>
<td>Returns the informat name that is associated with the specified variable</td>
</tr>
<tr>
<td></td>
<td>“VINFORMATNX Function” on page 899</td>
<td>Returns the informat name that is associated with the value of the specified argument</td>
</tr>
<tr>
<td></td>
<td>“VINFORMATW Function” on page 900</td>
<td>Returns the informat width that is associated with the specified variable</td>
</tr>
<tr>
<td></td>
<td>“VINFORMATWX Function” on page 901</td>
<td>Returns the informat width that is associated with the value of the specified argument</td>
</tr>
<tr>
<td></td>
<td>“VINFORMATX Function” on page 902</td>
<td>Returns the informat that is associated with the value of the specified argument</td>
</tr>
<tr>
<td></td>
<td>“VLABEL Function” on page 904</td>
<td>Returns the label that is associated with the specified variable</td>
</tr>
<tr>
<td></td>
<td>“VLABELX Function” on page 905</td>
<td>Returns the variable label for the value of the specified argument</td>
</tr>
<tr>
<td></td>
<td>“VLENGTH Function” on page 906</td>
<td>Returns the compile-time (allocated) size of the specified variable</td>
</tr>
<tr>
<td></td>
<td>“VLENGTHX Function” on page 907</td>
<td>Returns the compile-time (allocated) size for the value of the specified argument</td>
</tr>
<tr>
<td></td>
<td>“VNAME Function” on page 908</td>
<td>Returns the name of the specified variable</td>
</tr>
<tr>
<td></td>
<td>“VNAMEX Function” on page 909</td>
<td>Validates the value of the specified argument as a variable name</td>
</tr>
<tr>
<td></td>
<td>“VTRANSCODE Function” on page 910</td>
<td>Returns a value that indicates whether transcoding is on or off for the specified character variable</td>
</tr>
<tr>
<td></td>
<td>“VTRANSCODEX Function” on page 911</td>
<td>Returns a value that indicates whether transcoding is on or off for the specified argument</td>
</tr>
<tr>
<td>Category</td>
<td>Functions and CALL Routines</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Category Functions and CALL Routines</td>
<td>“VTYPE Function” on page 911</td>
<td>Returns the type (character or numeric) of the specified variable</td>
</tr>
<tr>
<td></td>
<td>“VTYPEX Function” on page 912</td>
<td>Returns the type (character or numeric) for the value of the specified argument</td>
</tr>
<tr>
<td></td>
<td>“VVALUE Function” on page 913</td>
<td>Returns the formatted value that is associated with the variable that you specify</td>
</tr>
<tr>
<td></td>
<td>“VVALUEX Function” on page 914</td>
<td>Returns the formatted value that is associated with the argument that you specify</td>
</tr>
<tr>
<td>Web Tools</td>
<td>“HTMLDECODE Function” on page 580</td>
<td>Decodes a string containing HTML numeric character references or HTML character entity references and returns the decoded string</td>
</tr>
<tr>
<td>Web Tools</td>
<td>“HTMLENCODE Function” on page 581</td>
<td>Encodes characters using HTML character entity references and returns the encoded string</td>
</tr>
<tr>
<td>Web Tools</td>
<td>“URLDECODE Function” on page 868</td>
<td>Returns a string that was decoded using the URL escape syntax</td>
</tr>
<tr>
<td>Web Tools</td>
<td>“URLENCODE Function” on page 869</td>
<td>Returns a string that was encoded using the URL escape syntax</td>
</tr>
</tbody>
</table>

## Dictionary

### ABS Function

Returns the absolute value

**Category:** Mathematical

#### Syntax

ABS (argument)

#### Arguments

*argument* is numeric.

#### Details

The ABS function returns a nonnegative number that is equal in magnitude to that of the argument.
Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=abs(2.4);</td>
<td>2.4</td>
</tr>
<tr>
<td>x=abs(-3);</td>
<td>3</td>
</tr>
</tbody>
</table>

**ADDR Function**

Returns the memory address of a numeric variable on a 32–bit platform

Category: Special

Restriction: Use on 32–bit platforms only.

Syntax

`ADDR(variable)`

Arguments

`variable`

specifies a variable name.

Details

The value that is returned is always numeric. Because the storage location of a variable can vary from one execution to the next, the value that is returned by ADDR can vary. The ADDR function is used mostly in combination with the PEEK and PEEKC functions and the CALL POKE routine.

You cannot use the ADDR function on 64-bit platforms. If you attempt to use it, SAS writes a message to the log stating that this restriction applies. If you have legacy applications that use ADDR, change the applications and use ADDRLONG instead. You can use ADDRLONG on both 32–bit and 64–bit platforms.

Comparisons

The ADDR function returns the memory address of a numerical variable on a 32–bit platform. ADDRLONG returns the memory address of a character variable on 32–bit and 64–bit platforms.

Note: SAS recommends that you use ADDRLONG instead of ADDR because ADDRLONG can be used on both 32–bit and 64–bit platforms.
**Examples**

The following example returns the address at which the variable FIRST is stored:

```plaintext
data numlist;
   first=3;
   x=addr(first);
run;
```

**See Also**

CALL Routine:

“CALL POKE Routine” on page 352

Functions:

“PEEK Function” on page 708

“PEEKC Function” on page 709

“ADDRLONG Function” on page 296

---

**ADDRLONG Function**

Returns the memory address of a character variable on 32-bit and 64-bit platforms

**Category:** Special

---

**Syntax**

`ADDRLONG(variable)`

**Arguments**

`variable`

specifies the variable name that points to the memory address.

**Details**

The return value is a character variable that contains the binary representation of the pointer. To display this value, use the `$HEXw.` format to convert the binary value to its hexadecimal equivalent.

*Note:* If you used the ADDR function and defined an instance of the pointer as numeric by using a LENGTH statement in your program, substituting ADDR for ADDRLONG will fail because ADDRLONG requires a character argument.

**Examples**

The following example returns the pointer address for the variable ITEM, and formats the value.
The following line is written to the SAS log:

```
48063B020202020
```
## Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=airy(2.0);</td>
<td>0.0349241304</td>
</tr>
<tr>
<td>x=airy(-2.0);</td>
<td>0.2274074282</td>
</tr>
</tbody>
</table>

### ANYALNUM Function

Searches a character string for an alphanumeric character and returns the first position at which it is found.

**Category:** Character

**Syntax**

\[
\text{ANYALNUM}(\text{string } <,\text{start}>)
\]

**Arguments**

- **string**
  - specifies the character expression to search.

- **start**
  - is an optional integer that specifies the position at which the search should start and the direction in which to search.

**Details**

The results of the ANYALNUM function depend directly on the translation table that is in effect (see “TRANTAB= System Option” on page 1618) and indirectly on the ENCODING and LOCALE system options.

The ANYALNUM function searches a string for the first occurrence of any character that is a digit or an uppercase or lowercase letter. If such a character is found, ANYALNUM returns the position in the string of that character. If no such character is found, ANYALNUM returns a value of 0.

If you use only one argument, ANYALNUM begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, `start`, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of `start` is positive, the search proceeds to the right.
- If the value of `start` is negative, the search proceeds to the left.
- If the value of `start` is less than the negative length of the string, the search begins at the end of the string.

ANYALNUM returns a value of zero when

- the character that you are searching for is not found
the value of start is greater than the length of the string
the value of start = 0.

Comparisons

The ANYALNUM function searches a character string for an alphanumeric character. The NOTALNUM function searches a character string for a non-alphanumeric character.

Examples

Example 1: Scanning a String from Left to Right

The following example uses the ANYALNUM function to search a string from left to right for alphanumeric characters.

```sas
data _null_
string='Next = Last + 1;'
j=0;
do until(j=0);
j=anyalnum(string,j+1);
if j=0 then put +3 "That's all";
else do;
c=substr(string,j,1);
put +3 j= c=
end;
end;
run;
```

The following lines are written to the SAS log:

```
j=1 c=N
j=2 c=e
j=3 c=x
j=4 c=t
j=8 c=L
j=9 c=a
j=10 c=s
j=11 c=t
j=15 c=1
That's all
```

Example 2: Scanning a String from Right to Left

The following example uses the ANYALNUM function to search a string from right to left for alphanumeric characters.

```sas
data _null_
string='Next = Last + 1;'
j=999999;
do until(j=0);
j=anyalnum(string,1-j);
if j=0 then put +3 "That's all";
else do;
c=substr(string,j,1);
put +3 j= c=
end;
end;
run;
```
The following lines are written to the SAS log:

```
j=15 c=1
j=11 c=t
j=10 c=s
j=9 c=a
j=8 c=L
j=4 c=t
j=3 c=x
j=2 c=e
j=1 c=N
That's all
```

See Also

Function:
“NOTALNUM Function” on page 665

---

**ANYALPHA Function**

Searches a character string for an alphabetic character and returns the first position at which it is found.

**Category:** Character

**Syntax**

\[ \text{ANYALPHA}(\text{string} \, <,\text{start}>) \]

**Arguments**

- **string**
  is the character constant, variable, or expression to search.

- **start**
  is an optional integer that specifies the position at which the search should start and the direction in which to search.

**Details**

The results of the ANYALPHA function depend directly on the translation table that is in effect (see “TRANTAB= System Option” on page 1618) and indirectly on the ENCODING and LOCALE system options.

The ANYALPHA function searches a string for the first occurrence of any character that is an uppercase or lowercase letter. If such a character is found, ANYALPHA returns the position in the string of that character. If no such character is found, ANYALPHA returns a value of 0.

If you use only one argument, ANYALPHA begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, \( \text{start} \),
specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of `start` is positive, the search proceeds to the right.
- If the value of `start` is negative, the search proceeds to the left.
- If the value of `start` is less than the negative length of the string, the search begins at the end of the string.

`ANYALPHA` returns a value of zero when
- the character that you are searching for is not found
- the value of `start` is greater than the length of the string
- the value of `start` = 0.

**Comparisons**

The `ANYALPHA` function searches a character string for an alphabetic character. The `NOTALPHA` function searches a character string for a non-alphabetic character.

**Examples**

The following example uses the `ANYALPHA` function to search a string from left to right for alphabetic characters.

```sas
data _null_;
  string='Next = _n_ + 12E3;';
  j=0;
  do until(j=0);
    j=anyalpha(string,j+1);
    if j=0 then put +3 "That’s all";
    else do;
      c=substr(string,j,1);
      put +3 j= c=;
    end;
  end;
run;
```

The following lines are written to the SAS log:

```
j=1 c=N
j=2 c=e
j=3 c=x
j=4 c=t
j=9 c=n
j=16 c=E
That’s all
```

**See Also**

Function:

“`NOTALPHA` Function” on page 666

---

**ANYCNTRL Function**

Searches a character string for a control character and returns the first position at which it is found
Category: Character

Syntax

ANYCNTRL(string <,start>)

Arguments

string
is the character constant, variable, or expression to search.

start
is an optional integer that specifies the position at which the search should start and the direction in which to search.

Details

The ANYCNTRL function searches a string for the first occurrence of a control character. If such a character is found, ANYCNTRL returns the position in the string of that character. If no such character is found, ANYCNTRL returns a value of 0.

If you use only one argument, ANYCNTRL begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, start, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of start is positive, the search proceeds to the right.
- If the value of start is negative, the search proceeds to the left.
- If the value of start is less than the negative length of the string, the search begins at the end of the string.

ANYCNTRL returns a value of zero when

- the character that you are searching for is not found
- the value of start is greater than the length of the string
- the value of start = 0.

Comparisons

The ANYCNTRL function searches a character string for a control character. The NOTCNTRL function searches a character string for a character that is not a control character.

See Also

Function:

“NOTCNTRL Function” on page 668

ANYDIGIT Function

Searches a character string for a digit and returns the first position at which it is found
ANYDIGIT Function 303

Syntax

ANYDIGIT(string <,start>)

Arguments

string

is the character constant, variable, or expression to search.

start

is an optional integer that specifies the position at which the search should start and the direction in which to search.

Details

The ANYDIGIT function searches a string for the first occurrence of any character that is a digit. If such a character is found, ANYDIGIT returns the position in the string of that character. If no such character is found, ANYDIGIT returns a value of 0.

If you use only one argument, ANYDIGIT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, start, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of start is positive, the search proceeds to the right.
- If the value of start is negative, the search proceeds to the left.
- If the value of start is less than the negative length of the string, the search begins at the end of the string.

ANYDIGIT returns a value of zero when

- the character that you are searching for is not found
- the value of start is greater than the length of the string
- the value of start = 0.

Comparisons

The ANYDIGIT function searches a character string for a digit. The NOTDIGIT function searches a character string for any character that is not a digit.

Examples

The following example uses the ANYDIGIT function to search for a character that is a digit.

```plaintext
data _null_
    string='Next = _n_ + 12E3;'
    j=0;
    do until(j=0);
        j=anydigit(string,j+1);
        if j=0 then put +3 "That's all";
        else do;
            c=substr(string,j,1);
```
The following lines are written to the SAS log:

```
j=14 c=1
j=15 c=2
j=17 c=3
That’s all
```

See Also

Function:
“NOTDIGIT Function” on page 669
specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of `start` is positive, the search proceeds to the right.
- If the value of `start` is negative, the search proceeds to the left.
- If the value of `start` is less than the negative length of the string, the search begins at the end of the string.

ANYFIRST returns a value of zero when

- the character that you are searching for is not found
- the value of `start` is greater than the length of the string
- the value of `start` = 0.

**Comparisons**

The ANYFIRST function searches a string for the first occurrence of any character that is valid as the first character in a SAS variable name under `VALIDVARNAMES=V7`. The NOTFIRST function searches a string for the first occurrence of any character that is not valid as the first character in a SAS variable name under `VALIDVARNAMES=V7`.

**Examples**

The following example uses the ANYFIRST function to search a string for any character that is valid as the first character in a SAS variable name under `VALIDVARNAMES=V7`.

```sas
data _null_
  string='Next = _n_ + 12E3;' j=0;
  do until(j=0);
    j=anyfirst(string,j+1);
    if j=0 then put +3 "That's all";
    else do;
      c=substr(string,j,1);
      put +3 j= c=;
    end;
  end;
r
run;
```

The following lines are written to the SAS log:

```
j=1 c=N
j=2 c=e
j=3 c=x
j=4 c=t
j=8 c=_
j=9 c=n
j=10 c=_
j=16 c=E
That's all
```

**See Also**

Function:

“NOTFIRST Function” on page 672
**ANYGRAPH Function**

Searches a character string for a graphical character and returns the first position at which it is found

**Category:** Character

**Syntax**

ANYGRAPH(string <,start>)

**Arguments**

*string*

is the character constant, variable, or expression to search.

*start*

is an optional integer that specifies the position at which the search should start and the direction in which to search.

**Details**

The results of the ANYGRAPH function depend directly on the translation table that is in effect (see “TRANTAB= System Option” on page 1618) and indirectly on the ENCODING and LOCALE system options.

The ANYGRAPH function searches a string for the first occurrence of a graphical character. A graphical character is defined as any printable character other than white space. If such a character is found, ANYGRAPH returns the position in the string of that character. If no such character is found, ANYGRAPH returns a value of 0.

If you use only one argument, ANYGRAPH begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYGRAPH returns a value of zero when

- the character that you are searching for is not found
- the value of *start* is greater than the length of the string
- the value of *start* = 0.

**Comparisons**

The ANYGRAPH function searches a character string for a graphical character. The NOTGRAPH function searches a character string for a non-graphical character.

**Examples**

The following example uses the ANYGRAPH function to search a string for graphical characters.
Functions and CALL Routines

ANYLOWER Function 307

data _null_
   string='Next = _n_ + 12E3;'
   j=0;
   do until(j=0);
      j=anygraph(string,j+1);
      if j=0 then put +3 "That’s all";
      else do;
         c=substr(string,j,1);
         put +3 j= c=;
      end;
   end;
run;

The following lines are written to the SAS log:
   j=1 c=N
   j=2 c=e
   j=3 c=x
   j=4 c=t
   j=6 c==
   j=8 c=_
   j=9 c=n
   j=10 c=_
   j=12 c=+
   j=14 c=1
   j=15 c=2
   j=16 c=E
   j=17 c=3
   j=18 c=;
   That’s all

See Also

Function:
   “NOTGRAPH Function” on page 674

---

**ANYLOWER Function**

Searches a character string for a lowercase letter and returns the first position at which it is found

**Category:** Character

**Syntax**

**ANYLOWER**(string <,start>)
Arguments

string
is the character constant, variable, or expression to search.

start
is an optional integer that specifies the position at which the search should start and the direction in which to search.

Details

The results of the ANYLOWER function depend directly on the translation table that is in effect (see “TRANTAB= System Option” on page 1618) and indirectly on the ENCODING and LOCALE system options.

The ANYLOWER function searches a string for the first occurrence of a lowercase letter. If such a character is found, ANYLOWER returns the position in the string of that character. If no such character is found, ANYLOWER returns a value of 0.

If you use only one argument, ANYLOWER begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, start, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of start is positive, the search proceeds to the right.
- If the value of start is negative, the search proceeds to the left.
- If the value of start is less than the negative length of the string, the search begins at the end of the string.

ANYLOWER returns a value of zero when
- the character that you are searching for is not found
- the value of start is greater than the length of the string
- the value of start = 0.

Comparisons

The ANYLOWER function searches a character string for a lowercase letter. The NOTLOWER function searches a character string for a character that is not a lowercase letter.

Examples

The following example uses the ANYLOWER function to search a string for any character that is a lowercase letter.

data _null_
  string='Next = _n_ + 12E3;';
  j=0;
  do until(j=0);
      j=anylower(string,j+1);
      if j=0 then put +3 "That’s all";
      else do;
          c=substr(string,j,1);
          put +3 j= c=;
      end;
  end;
run;
The following lines are written to the SAS log:

```
j=2  c=e
j=3  c=x
j=4  c=t
j=9  c=n
That’s all
```

See Also

Function:
“NOTLOWER Function” on page 676

---

**ANYNAME Function**

Searches a character string for a character that is valid in a SAS variable name under VALIDVARNAME=V7, and returns the first position at which that character is found.

**Category:** Character

**Syntax**

```
ANYNAME(string <,start>)
```

**Arguments**

**string**
- Is the character constant, variable, or expression to search.

**start**
- Is an optional integer that specifies the position at which the search should start and the direction in which to search.

**Details**

The ANYNAME function does not depend on the TRANTAB, ENCODING, or LOCALE options.

The ANYNAME function searches a string for the first occurrence of any character that is valid in a SAS variable name under VALIDVARNAME=V7. These characters are the underscore (_), digits, and uppercase or lowercase English letters. If such a character is found, ANYNAME returns the position in the string of that character. If no such character is found, ANYNAME returns a value of 0.

If you use only one argument, ANYNAME begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, `start`, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of `start` is positive, the search proceeds to the right.
- If the value of `start` is negative, the search proceeds to the left.
If the value of `start` is less than the negative length of the string, the search begins at the end of the string.

`ANYNAME` returns a value of zero when

- the character that you are searching for is not found
- the value of `start` is greater than the length of the string
- the value of `start` = 0.

**Comparisons**

The `ANYNAME` function searches a string for the first occurrence of any character that is valid in a SAS variable name under `VALIDVARNAME=V7`. The `NOTNAME` function searches a string for the first occurrence of any character that is not valid in a SAS variable name under `VALIDVARNAME=V7`.

**Examples**

The following example uses the `ANYNAME` function to search a string for any character that is valid in a SAS variable name under `VALIDVARNAME=V7`.

```sas
data _null_;  
string='Next = _n_ + 12E3;';  
j=0;  
do until(j=0);  
j=anyname(string,j+1);  
  if j=0 then put +3 "That's all";  
  else do;  
    c=substr(string,j,1);  
    put +3 j= c=;  
  end;  
end;  
run;
```

The following lines are written to the SAS log:

```
j=1  c=N  
j=2  c=e  
j=3  c=x  
j=4  c=t  
j=8  c=_  
j=9  c=n  
j=10 c=_  
j=14 c=l  
j=15 c=2  
j=16 c=E  
j=17 c=3  
That’s all
```

**See Also**

Function:

“`NOTNAME Function`” on page 677
ANYPRINT Function

Searches a character string for a printable character and returns the first position at which it is found.

Category: Character

Syntax

ANYPRINT(string <,start>)

Arguments

string
  is the character constant, variable, or expression to search.

start
  is an optional integer that specifies the position at which the search should start and the direction in which to search.

Details

The results of the ANYPRINT function depend directly on the translation table that is in effect (see “TRANTAB= System Option” on page 1618) and indirectly on the ENCODING and LOCALE system options.

The ANYPRINT function searches a string for the first occurrence of a printable character. If such a character is found, ANYPRINT returns the position in the string of that character. If no such character is found, ANYPRINT returns a value of 0.

If you use only one argument, ANYPRINT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, start, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of start is positive, the search proceeds to the right.
- If the value of start is negative, the search proceeds to the left.
- If the value of start is less than the negative length of the string, the search begins at the end of the string.

ANYPRINT returns a value of zero when

- the character that you are searching for is not found
- the value of start is greater than the length of the string
- the value of start = 0.

Comparisons

The ANYPRINT function searches a character string for a printable character. The NOTPRINT function searches a character string for a non-printable character.
Examples

The following example uses the ANYPRINT function to search a string for printable characters.

```sas
data _null_;
  string='Next = _n_ + 12E3;';
  j=0;
  do until(j=0);
    j=anyprint(string,j+1);
    if j=0 then put +3 "That's all";
    else do;
      c=substr(string,j,1);
      put +3 j= c=;
    end;
  end;
run;
```

The following lines are written to the SAS log:

```
  j=1 c=N
  j=2 c=e
  j=3 c=x
  j=4 c=t
  j=5 c=
  j=6 c==
  j=7 c=
  j=8 c=_
  j=9 c=n
  j=10 c=_
  j=11 c=
  j=12 c=+
  j=13 c=
  j=14 c=1
  j=15 c=2
  j=16 c=E
  j=17 c=3
  j=18 c=;
  That's all
```

See Also

Function:

“NOTPRINT Function” on page 679

**ANYPUNCT Function**

Searches a character string for a punctuation character and returns the first position at which it is found

**Category:** Character

**See:** ANYPUNCT Function in the documentation for your operating environment.
Syntax

ANYPUNCT(string <,start>)

Arguments

string  
is the character constant, variable, or expression to search.

start  
is an optional integer that specifies the position at which the search should start and the direction in which to search.

Details

The results of the ANYPUNCT function depend directly on the translation table that is in effect (see “TRANTAB= System Option” on page 1618) and indirectly on the ENCODING and LOCALE system options.

The ANYPUNCT function searches a string for the first occurrence of a punctuation character. If such a character is found, ANYPUNCT returns the position in the string of that character. If no such character is found, ANYPUNCT returns a value of 0.

If you use only one argument, ANYPUNCT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, start, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of start is positive, the search proceeds to the right.
- If the value of start is negative, the search proceeds to the left.
- If the value of start is less than the negative length of the string, the search begins at the end of the string.

ANYPUNCT returns a value of zero when

- the character that you are searching for is not found
- the value of start is greater than the length of the string
- the value of start = 0.

Comparisons

The ANYPUNCT function searches a character string for a punctuation character. The NOTPUNCT function searches a character string for a character that is not a punctuation character.

Examples

The following example uses the ANYPUNCT function to search a string for punctuation characters.

```sas
data _null_
  string='Next = _n_ + 12E3;';
  j=0;
  do until(j=0);
    j=anypunct(string,j+1);
```

if j=0 then put +3 "That's all";
else do;
  c=substr(string,j,1);
  put +3 j= c=;
end;
end;
run;

The following lines are written to the SAS log:

j=6 c==
j=8 c=_
j=10 c=_
j=12 c=+
j=18 c=;
That's all

See Also

Function:

“NOTPUNCT Function” on page 680

ANYSPACE Function

Searches a character string for a white-space character (blank, horizontal and vertical tab, carriage return, line feed, form feed) and returns the first position at which it is found

Category: Character

Syntax

ANYSPACE(string <,start>)

Arguments

string
  is the character constant, variable, or expression to search.

start
  is an optional integer that specifies the position at which the search should start and the direction in which to search.

Details

The results of the ANYSPACE function depend directly on the translation table that is in effect (see “TRANTAB= System Option” on page 1618) and indirectly on the ENCODING and LOCALE system options.

The ANYSPACE function searches a string for the first occurrence of any character that is a blank, horizontal tab, vertical tab, carriage return, line feed, or form feed. If
such a character is found, ANYSPACE returns the position in the string of that character. If no such character is found, ANYSPACE returns a value of 0.

If you use only one argument, ANYSPACE begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, start, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of start is positive, the search proceeds to the right.
- If the value of start is negative, the search proceeds to the left.
- If the value of start is less than the negative length of the string, the search begins at the end of the string.

ANYSPACE returns a value of zero when
- the character that you are searching for is not found
- the value of start is greater than the length of the string
- the value of start = 0.

Comparisons

The ANYSPACE function searches a character string for the first occurrence of a character that is a blank, horizontal tab, vertical tab, carriage return, line feed, or form feed. The NOTSPACE function searches a character string for the first occurrence of a character that is not a blank, horizontal tab, vertical tab, carriage return, line feed, or form feed.

Examples

The following example uses the ANYSPACE function to search a string for a character that is a white-space character.

```sas
data _null_;  
  string='Next = _n_ + 12E3;';  
  j=0;  
  do until(j=0);  
    j=anyspace(string,j+1);  
    if j=0 then put +3 "That's all";  
    else do;  
      c=substr(string,j,1);  
      put +3 j= c=;  
    end;  
  end;  
run;
```

The following lines are written to the SAS log:

```
j=5 c=  
j=7 c=  
j=11 c=  
j=13 c=  
That's all
```

See Also

Function:

“NOTSPACE Function” on page 682
ANYUPPER Function

Searches a character string for an uppercase letter and returns the first position at which it is found

Category: Character

Syntax

\[
\text{ANYUPPER}(\text{string } <,\text{start}>)
\]

Arguments

\textit{string}  
is the character constant, variable, or expression to search.

\textit{start}  
is an optional integer that specifies the position at which the search should start and the direction in which to search.

Details

The results of the ANYUPPER function depend directly on the translation table that is in effect (see “TRANTAB= System Option” on page 1618) and indirectly on the ENCODING and LOCALE system options.

The ANYUPPER function searches a string for the first occurrence of an uppercase letter. If such a character is found, ANYUPPER returns the position in the string of that character. If no such character is found, ANYUPPER returns a value of 0.

If you use only one argument, ANYUPPER begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, \textit{start}, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of \textit{start} is positive, the search proceeds to the right.
- If the value of \textit{start} is negative, the search proceeds to the left.
- If the value of \textit{start} is less than the negative length of the string, the search begins at the end of the string.

ANYUPPER returns a value of zero when

- the character that you are searching for is not found
- the value of \textit{start} is greater than the length of the string
- the value of \textit{start} = 0.

Comparisons

The ANYUPPER function searches a character string for an uppercase letter. The NOTUPPER function searches a character string for a character that is not an uppercase letter.

Examples

The following example uses the ANYUPPER function to search a string for an uppercase letter.
data _null_
  string='Next = _n_ + 12E3;';
  j=0;
  do until(j=0);
    j=anyupper(string,j+1);
    if j=0 then put +3 "That's all";
    else do;
      c=substr(string,j,1);
      put +3 j= c=;
    end;
  end;
run;

The following lines are written to the SAS log:

j=1 c=N
j=16 c=E
That's all

See Also

Function:
  “NOTUPPER Function” on page 684

---

ANYXDIGIT Function

Searches a character string for a hexadecimal character that represents a digit and returns the first position at which that character is found

Category: Character

Syntax

ANYXDIGIT(string,<,start>)

Arguments

string
  is the character constant, variable, or expression to search.

start
  is an optional integer that specifies the position at which the search should start and the direction in which to search.

Details

The ANYXDIGIT function searches a string for the first occurrence of any character that is a digit or an uppercase or lowercase A, B, C, D, E, or F. If such a character is
found, ANYXDIGIT returns the position in the string of that character. If no such character is found, ANYXDIGIT returns a value of 0.

If you use only one argument, ANYXDIGIT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, start, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of start is positive, the search proceeds to the right.
- If the value of start is negative, the search proceeds to the left.
- If the value of start is less than the negative length of the string, the search begins at the end of the string.

ANYXDIGIT returns a value of zero when

- the character that you are searching for is not found
- the value of start is greater than the length of the string
- the value of start = 0.

**Comparisons**

The ANYXDIGIT function searches a character string for a character that is a hexadecimal digit. The NOTXDIGIT function searches a character string for a character that is not a hexadecimal digit.

**Examples**

The following example uses the ANYXDIGIT function to search a string for a hexadecimal character that represents a digit.

```sas
data _null_;
  string='Next = _n_ + 12E3;';
  j=0;
  do until(j=0);
    j=anyxdigit(string,j+1);
    if j=0 then put +3 "That’s all";
    else do;
      c=substr(string,j,1);
      put +3 j= c=;
    end;
  end;
run;
```

The following lines are written to the SAS log:

```
j=2  c=e
j=14 c=1
j=15 c=2
j=16 c=E
j=17 c=3
That’s all
```

**See Also**

Function:

“NOTXDIGIT Function” on page 685
**ARCOS Function**

Returns the arccosine

Category: Trigonometric

**Syntax**

ARCOS (argument)

**Arguments**

*argument* is numeric.

**Details**

The ARCOS function returns the arccosine (inverse cosine) of the argument. The value returned is in radians.

**Examples**

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=arcos(1);</td>
<td>0</td>
</tr>
<tr>
<td>x=arcos(0);</td>
<td>1.5707963268</td>
</tr>
<tr>
<td>x=arcos(-0.5);</td>
<td>2.0943951024</td>
</tr>
</tbody>
</table>

---

**ARSIN Function**

Returns the arcsine

Category: Trigonometric

**Syntax**

ARSIN (argument)
Arguments

\textit{argument} is numeric.

\textbf{Range:} between $-1$ and 1

Details

The ARSIN function returns the arcsine (inverse sine) of the argument. The value returned is in radians.

Examples

\begin{tabular}{ll}
\textbf{SAS Statements} & \textbf{Results} \\
\hline
\texttt{x=arsin(0);} & 0 \\
\texttt{x=arsin(1);} & 1.5707963268 \\
\texttt{x=arsin(-0.5);} & -0.523598776 \\
\end{tabular}

\textbf{ATAN Function}

\textbf{Returns the arc tangent}

\textbf{Category:} Trigonometric

Syntax

\texttt{ATAN (argument)}

Arguments

\textit{argument} is numeric.

Details

The ATAN function returns the 2-quadrant arc tangent (inverse tangent) of the argument. The value that is returned is the angle (in radians) whose tangent is $x$ and whose value ranges from $-\pi/2$ to $\pi/2$. If the argument is missing, then ATAN returns a missing value.
Comparisons

The ATAN function is similar to the ATAN2 function except that ATAN2 calculates the arc tangent of the angle from the values of two arguments rather than from one argument.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=atan(0);</td>
<td>0</td>
</tr>
<tr>
<td>x=atan(1);</td>
<td>0.7853981634</td>
</tr>
<tr>
<td>x=atan(-9.0);</td>
<td>-1.460139106</td>
</tr>
</tbody>
</table>

See Also

“ATAN2 Function” on page 321

ATAN2 Function

Returns the arc tangent of two numeric variables

Category: Trigonometric

Syntax

ATAN2(argument-1, argument-2)

Arguments

argument-1

is numeric.

argument-2

is numeric.

Details

The ATAN2 function returns the arc tangent (inverse tangent) of two numeric variables. The result of this function is similar to the result of calculating the arc tangent of argument-1 / argument-2, except that the signs of both arguments are used to determine the quadrant of the result. ATAN2 returns the result in radians, which is a value between -π and π. If either of the arguments in ATAN2 is missing, then ATAN2 returns a missing value.
Comparisons

The ATAN2 function is similar to the ATAN function except that ATAN calculates the arc tangent of the angle from the value of one argument rather than from two arguments.

Examples

<table>
<thead>
<tr>
<th>SAS statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=atan2(-1, 0.5);</td>
<td>-1.107148718</td>
</tr>
<tr>
<td>b=atan2(6, 8);</td>
<td>0.6435011088</td>
</tr>
<tr>
<td>c=atan2(5, -3);</td>
<td>2.1112158271</td>
</tr>
</tbody>
</table>

See Also

Functions:

“ATAN Function” on page 320

ATTRC Function

Returns the value of a character attribute for a SAS data set

Category: SAS File I/O

Syntax

ATTRC(data-set-id, attr-name)

Arguments

data-set-id
specifies the data set identifier that the OPEN function returns.

attr-name
is an attribute name. If attr-name is invalid, a missing value is returned.

Valid values for use with attr-name are:

CHARSET
returns a value for the character set of the machine that created the data set.

eempty string       Data set not sorted
ASCII               ASCII character set
ENCRYPT
returns 'YES' or 'NO' depending on whether the SAS data set is encrypted.

ENGINE
returns the name of the engine that is used to access the data set.

LABEL
returns the label assigned to the data set.

LIB
returns the libref of the SAS data library in which the data set resides.

MEM
returns the SAS data set name.

MODE
returns the mode in which the SAS data set was opened, such as:
I INPUT mode allows random access if the engine supports it; otherwise, it defaults to IN mode.
IN INPUT mode reads sequentially and allows revisiting observations.
IS INPUT mode reads sequentially but does not allow revisiting observations.
N NEW mode creates a new data set.
U UPDATE mode allows random access if the engine supports it; otherwise, it defaults to UN mode.
UN UPDATE mode reads sequentially and allows revisiting observations.
US UPDATE mode reads sequentially but does not allow revisiting observations.
V UTILITY mode allows modification of variable attributes and indexes associated with the data set.

MTYPE
returns the SAS data library member type.

SORTEDBY
returns an empty string if the data set is not sorted. Otherwise, it returns the names of the BY variables in the standard BY statement format.

SORTLVL
returns a value that indicates how a data set was sorted:
Empty string Data set is not sorted.
WEAK Sort order of the data set was established by the user (for example, through the SORTEDBY data set option). The system cannot validate its correctness, so the order of observations cannot be depended on.
STRONG Sort order of the data set was established by the software (for example, through PROC SORT or the OUT= option in the CONTENTS procedure).
SORTSEQ returns an empty string if the data set is sorted on the native machine or if the sort collating sequence is the default for the operating environment. Otherwise, it returns the name of the alternate collating sequence used to sort the file.

TYPE returns the SAS data set type.

Examples

□ This example generates a message if the SAS data set has not been opened in INPUT SEQUENTIAL mode. The message is written to the SAS log as follows:

```sas
%let mode=%sysfunc(attrc(&dsid,MODE));
%if &mode ne IS %then
   %put Data set has not been opened in INPUT SEQUENTIAL mode.;
```

□ This example tests whether a data set has been sorted and writes the result to the SAS log.

```sas
data _null_; 
   dsid=open("sasdata.sortcars","i");
   charset=attrc(dsid,"CHARSET");
   if charset = "" then
      put "Data set has not been sorted.";
   else put "Data set sorted with " charset "character set.";
   rc=close(dsid);
run;
```

See Also

Functions:

“ATTRN Function” on page 324
“OPEN Function” on page 690

ATTRN Function

Returns the value of a numeric attribute for the specified SAS data set

Category: SAS File I/O

Syntax

`ATTRN(data-set-id,attr-name)`
Arguments

**data-set-id**
specifies the data set identifier that the OPEN function returns.

**attr-name**
is a numeric attribute, as listed in the section below. If the value of attr-name is invalid, a missing value is returned.

Valid numeric values used with attr-name are:

**ALTERPW**
specifies whether a password is required to alter the data set.
1 the data set is alter protected.
0 the data set is not alter protected.

**ANOBS**
specifies whether the engine knows the number of observations.
1 the engine knows the number of observations.
0 the engine does not know the number of observations.

**ANY**
specifies whether the data set has observations or variables.
−1 the data set has no observations or variables.
0 the data set has no observations.
1 the data set has observations and variables.

Alias: VAROBS

**ARAND**
specifies whether the engine supports random access.
1 the engine supports random access.
0 the engine does not support random access.

Alias: RANDOM

**ARWU**
specifies whether the engine can manipulate files.
1 the engine is not read-only. It can create or update SAS files.
0 the engine is read-only.

**AUDIT**
specifies whether logging to an audit file is enabled.
1 logging is enabled.
0 logging is suspended.

**AUDIT_DATA**
specifies whether after-update record images are stored.
1 after-update record images are stored.
0 after-update record images are not stored.
**AUDIT_BEFORE**
specifies whether before-update record images are stored.
- 1: before-update record images are stored.
- 0: before-update record images are not stored.

**AUDIT_ERROR**
specifies whether unsuccessful after-update record images are stored.
- 1: unsuccessful after-update record images are stored.
- 0: unsuccessful after-update record images are not stored.

**CRDTE**
specifies the date that the data set was created. The value that is returned is the internal SAS datetime value for the creation date.

**Tip:** Use the DATETIME format to display this value.

**ICONST**
returns information about the existence of integrity constraints for a SAS data set.
- 0: no integrity constraints.
- 1: one or more general integrity constraints.
- 2: one or more referential integrity constraints.
- 3: both one or more general integrity constraints and one or more referential integrity constraints.

**INDEX**
specifies whether the data set supports indexing.
- 1: indexing is supported.
- 0: indexing is not supported.

**ISINDEX**
specifies whether the data set is indexed.
- 1: at least one index exists for the data set.
- 0: the data set is not indexed.

**ISSUBSET**
specifies whether the data set is a subset.
- 1: at least one WHERE clause is active.
- 0: no WHERE clause is active.

**LRECL**
specifies the logical record length.

**LRID**
specifies the length of the record ID.

**MAXGEN**
specifies the maximum number of generations.

**MAXRC**
specifies whether an application checks return codes.
- 1: an application checks return codes.
- 0: an application does not check return codes.
MODTE
 specifies the last date and time that the data set was modified. The value returned is
 the internal SAS datetime value.
Tip: Use the DATETIME. format to display this value.

NDEL
 specifies the number of observations in the data set that are marked for deletion.

NEXTGEN
 specifies the next generation number to generate.

NLOBS
 specifies the number of logical observations (those that are not marked for deletion).
An active WHERE clause does not affect this number.
−1 the number of observations is not available.

NLOBSF
 specifies the number of logical observations (those that are not marked for deletion)
by forcing each observation to be read and by taking the FIRSTOBS system option,
the OBS system option, and the WHERE clauses into account.
Tip: Passing NLOBSF to ATTRN requires the engine to read every observation
 from the data set that matches the WHERE clause. Based on the file type and
 size, this can be a time-consuming process.

NOBS
 specifies the number of physical observations (including those that are marked for
deletion). An active WHERE clause does not affect this number.
−1 the number of observations is not available.

NVARS
 specifies the number of variables in the data set.

PW
 specifies whether a password is required to access the data set.
1 the data set is protected.
0 the data set is not protected.

RADIX
 specifies whether access by observation number (radix addressability) is allowed.
1 access by observation number is allowed.
0 access by observation number is not allowed.
Note: A data set that is accessed by a tape engine is index addressable although it
 cannot be accessed by an observation number.

READPW
 specifies whether a password is required to read the data set.
1 the data set is read protected.
0 the data set is not read protected.

TAPE
 specifies the status of the data set tape.
1 the data set is a sequential file.
0 the data set is not a sequential file.
WHSTMT
specifies the active WHERE clauses.
0 no WHERE clause is active.
1 a permanent WHERE clause is active.
2 a temporary WHERE clause is active.
3 both permanent and temporary WHERE clauses are active.

WRITEPW
specifies whether a password is required to write to the data set.
1 the data set is write protected.
0 the data set is not write protected.

Examples

☐ This example checks whether a WHERE clause is currently active for a data set.

```sas
%let iswhere=%sysfunc(attrn(&dsid,whstmt));
%if &iswhere %then
  %put A WHERE clause is currently active.;
```

☐ This example checks whether a data set is protected with a password.

```sas
data _null_;
  dsid=open("mydata");
  pw=attrn(dsid,"pw");
  if pw then put "data set is protected";
run;
```

See Also

Functions:
“ATTRC Function” on page 322
“OPEN Function” on page 690

---

**BAND Function**

Returns the bitwise logical AND of two arguments

Category: Bitwise Logical Operations

**Syntax**

```
bang(argument-1,argument-2)
```
Arguments

\textit{argument-1, argument-2} are numeric, nonnegative, and nonmissing. Separate the arguments with a comma.

\textbf{Range:} 0 to the largest 32-bit unsigned integer

Examples

\begin{center}
\begin{tabular}{ll}
SAS Statements & Results \\
\hline
x=b\texttt{and}(0Fx,05x); & \\
\text{put } x=\texttt{hex.}; & \texttt{x=00000005;} \\
\end{tabular}
\end{center}

\textbf{BETA Function}

Returns the value of the beta function

\textbf{Category:} Mathematical

\textbf{Syntax}

BETA\texttt{(}a,b\texttt{)}

\textbf{Arguments}

\begin{itemize}
\item \texttt{a} is the first shape parameter, where \(a>0\).
\item \texttt{b} is the second shape parameter, where \(b>0\).
\end{itemize}

\textbf{Details}

The BETA function is mathematically given by the equation

\[ \beta(a, b) = \int_0^1 x^{a-1} (1-x)^{b-1} \, dx \]

with \(a>0, b>0\). It should be noted that

\[ \beta(a, b) = \frac{\Gamma(a) \Gamma(b)}{\Gamma(a + b)} \]
where \( \Gamma(.) \) is the gamma function. If the expression cannot be computed, BETA returns a missing value.

**Examples**

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=beta(5,3);</td>
<td>0.9523809524e-2</td>
</tr>
</tbody>
</table>

**See Also**

Function:

“LOGBETA Function” on page 635

**BETAINV Function**

Returns a quantile from the beta distribution

**Category:** Quantile

**Syntax**

BETAINV \((p,a,b)\)

**Arguments**

- \( p \) is a numeric probability.
  - Range: \( 0 \leq p \leq 1 \)
- \( a \) is a numeric shape parameter.
  - Range: \( a > 0 \)
- \( b \) is a numeric shape parameter.
  - Range: \( b > 0 \)

**Details**

The BETAINV function returns the \( p \)th quantile from the beta distribution with shape parameters \( a \) and \( b \). The probability that an observation from a beta distribution is less than or equal to the returned quantile is \( p \).
Note: BETAINV is the inverse of the PROBBETA function.

**Examples**

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x=betainv(0.001,2,4);</code></td>
<td><code>0.0101017879</code></td>
</tr>
</tbody>
</table>

**BLSHIFT Function**

Returns the bitwise logical left shift of two arguments

Category: Bitwise Logical Operations

**Syntax**

`BLSHIFT(argument-1,argument-2)`

**Arguments**

- **argument-1**
  - is numeric, nonnegative, and nonmissing.
  - **Range:** 0 to the largest 32-bit unsigned integer

- **argument-2**
  - is numeric, nonnegative, and nonmissing.
  - **Range:** 0 to 31, inclusive

**Examples**

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x=blshift(07x,2);</code></td>
<td></td>
</tr>
<tr>
<td>`put x=hex.;</td>
<td><code>x=0000001C</code></td>
</tr>
</tbody>
</table>

**BNOT Function**

Returns the bitwise logical NOT of an argument

Category: Bitwise Logical Operations
### Syntax

BNOT(argument)

### Arguments

**argument**
- is numeric, nonnegative, and nonmissing.
  - **Range:** 0 to the largest 32-bit unsigned integer

### Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=bnot(0F000000Fx); put x=hex.;</td>
<td>x=0FFFFFF0</td>
</tr>
</tbody>
</table>

### BOR Function

Returns the bitwise logical OR of two arguments

**Category:** Bitwise Logical Operations

### Syntax

BOR(argument-1,argument-2)

### Arguments

**argument-1,argument-2**
- are numeric, nonnegative, and nonmissing. Separate the arguments with a comma.
  - **Range:** 0 to the largest 32-bit unsigned integer

### Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=bor(01x,0F4x); put x=hex.;</td>
<td>x=000000F5</td>
</tr>
</tbody>
</table>
BRSHIFT Function

Returns the bitwise logical right shift of two arguments

Category:  Bitwise Logical Operations

Syntax

BRSHIFT(argument-1, argument-2)

Arguments

argument-1
  is numeric, nonnegative, and nonmissing.
  Range:  0 to the largest 32-bit unsigned integer

argument-2
  is numeric, nonnegative, and nonmissing.
  Range:  0 to 31, inclusive

Examples

SAS Statements Results
x=brshift(01Cx,2);
put x=hex.;  x=00000007

BXOR Function

Returns the bitwise logical EXCLUSIVE OR of two arguments

Category:  Bitwise Logical Operations

Syntax

BXOR(argument-1, argument-2)
Arguments

_argument-1, argument-2_
are numeric, nonnegative, and nonmissing. Separate the arguments with a comma.

**Range:** 0 to the largest 32-bit unsigned integer

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=bxor(03x,01x); put x=hex.;</td>
<td>x=00000002</td>
</tr>
</tbody>
</table>

BYTE Function

Returns one character in the ASCII or the EBCDIC collating sequence

**Category:** Character

**See:** BYTE Function in the documentation for your operating environment.

**Syntax**

BYTE \((n)\)

**Arguments**

\(n\)

specifies an integer that represents a specific ASCII or EBCDIC character.

**Range:** 0–255

**Details**

If the BYTE function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 1.

For EBCDIC collating sequences, \(n\) is between 0 and 255. For ASCII collating sequences, the characters that correspond to values between 0 and 127 represent the standard character set. Other ASCII characters that correspond to values between 128 and 255 are available on certain ASCII operating environments, but the information those characters represent varies with the operating environment.
Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ASCII</td>
</tr>
<tr>
<td></td>
<td>-------</td>
</tr>
<tr>
<td>x=byte(80);</td>
<td></td>
</tr>
<tr>
<td>put x;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>P</td>
</tr>
</tbody>
</table>

See Also

Functions:
“COLLATE Function” on page 440
“RANK Function” on page 780

CALL ALLPERM Routine

Generates all permutations of the values of several variables

Category: Mathematical

Syntax

CALL ALLPERM(k, variable-1<, variable-2, ...>);

Arguments

k
specifies an integer value that ranges from one to the number of permutations.

variable
specifies all numeric variables or all character variables that have the same length. The values of these variables are permuted.

Requirement: Initialize these variables before you call the ALLPERM routine.

Restriction: Specify no more than 18 variables.

Details

Use the CALL ALLPERM routine in a loop where the first argument accepts each integral value from one to the number of permutations. On the first call, the argument types and lengths are checked for consistency. On each subsequent call, the values of two of the variables are interchanged. Because each permutation is generated from the previous permutation by a single interchange, the algorithm is very efficient.
Note: You can compute the number of permutations by using the FACT function. See “FACT Function” on page 516 for more information.

If you use the CALL ALLPERM routine and the first argument is out of sequence, the results are not useful.

In particular, if you initialize the variables and then immediately call the ALLPERM routine with a first argument of \( k \), for example, your result will not be the \( k \)th permutation (except when \( k \) is one). To get the \( k \)th permutation, you must call the ALLPERM routine \( k \) times, with the first argument having values from 1 through \( k \) in that exact order.

Examples

The following example generates permutations of given values by using the CALL ALLPERM routine.

```sas
data _null_
    array x [4] $3 ('ant' 'bee' 'cat' 'dog');
    n=dim(x);
    nfact=fact(n);
    do i=1 to nfact;
        call allperm(i, of x[*]);
        put i 5. +2 x[*];
    end;
run;
```

The following lines are written to the SAS log:

```
1 ant bee cat dog
2 ant bee dog cat
3 ant dog bee cat
4 dog ant bee cat
5 dog ant cat bee
6 ant dog cat bee
7 ant cat dog bee
8 ant cat bee dog
9 cat ant bee dog
10 cat ant dog bee
11 cat dog ant bee
12 dog cat ant bee
13 dog cat bee ant
14 cat dog bee ant
15 cat bee dog ant
16 cat bee ant dog
17 bee cat ant dog
18 bee cat dog ant
19 bee dog cat ant
20 dog bee cat ant
21 dog bee ant cat
22 bee dog ant cat
23 bee ant dog cat
24 bee ant cat dog
```
CALL CATS Routine

Concatenates character strings and removes leading and trailing blanks

Category: Character

Syntax

CALL CATS(result <, string-1, ...string-n>);

Arguments

result
specifies a SAS variable.

string
specifies a SAS character string.

Details

The CALL CATS routine returns the result in the first argument, result. The routine appends the values of the arguments that follow to result. If the length of result is not large enough to contain the entire result, SAS

- writes a warning message to the log stating that the result was truncated
- writes a note to the log that shows the location of the function call and lists the argument that caused the truncation, except in SQL or in a WHERE clause
- sets _ERROR_ to 1 in the DATA step, except in a WHERE clause.

The CALL CATS routine removes leading and trailing blanks from numeric arguments after it formats the numeric value with the BEST. format.

Comparisons

The results of the CALL CATS, CALL CATT, and CALL CATX routines are usually equivalent to statements that use the concatenation operator (||) and the TRIM and LEFT functions. However, using the CALL CATS, CALL CATT, and CALL CATX routines is faster than using TRIM and LEFT.

The following table shows statements that are equivalent to CALL CATS, CALL CATT, and CALL CATX. The variables X1 through X4 specify character variables, and SP specifies a separator, such as a blank or comma.
CALL Routine | Equivalent Statement
---|---
CALL CATS(res, OF X1-X4); | res = TRIM(LEFT(X1)) || TRIM(LEFT(X2)) || TRIM(LEFT(X3)) || TRIM(LEFT(X4));

CALL CATT(OF X1-X4); | X1 = TRIM(X1) || TRIM(X2) || TRIM(X3) || TRIM(X4);

CALL CATX(SP, OF X1-X4); | X1 = TRIM(LEFT(X1)) || SP || TRIM(LEFT(X2)) || SP || TRIM(LEFT(X3)) || SP || TRIM(LEFT(X4));

* If any of the arguments is blank, the results that are produced by CALL CATX differ slightly from the results that are produced by the concatenated code. In this case, CALL CATX omits the corresponding separator. For example, CALL CATX("+","X","","Z","") produces X+Z.

**Examples**

The following example shows how the CALL CATS routine concatenates strings.

```sas
data _null_
  length answer $ 36;
  x='Athens is t'
  y='he Olym'
  z='pic site for 2004.'
  call cats(answer,x,y,z);
  put answer;
run;
```

The following line is written to the SAS log:

```
---+---1---+---2---+---3---+---4---+---5---+---6---+---7
Athens is the Olympic site for 2004.
```

**See Also**

Functions and CALL Routines:
  “CAT Function” on page 408
  “CATS Function” on page 410
  “CATT Function” on page 412
  “CATX Function” on page 414
  “CALL CATT Routine” on page 338
  “CALL CATX Routine” on page 340

---

**CALL CATT Routine**

*Concatenates character strings and removes trailing blanks*

**Category:** Character
Syntax
CALL CATT(result <, string-1, ...string-n>);

Arguments

result
specifies a SAS variable.

string
specifies a SAS character string.

Details
The CALL CATT routine returns the result in the first argument, result. The routine appends the values of the arguments that follow to result. If the length of result is not large enough to contain the entire result, SAS

- writes a warning message to the log stating that the result was truncated
- writes a note to the log that shows the location of the function call and lists the argument that caused the truncation, except in SQL or in a WHERE clause
- sets _ERROR_ to 1 in the DATA step, except in a WHERE clause.

The CALL CATT routine removes leading and trailing blanks from numeric arguments after it formats the numeric value with the BEST. format.

Comparisons
The results of the CALL CATS, CALL CATT, and CALL CATX routines are usually equivalent to statements that use the concatenation operator (||) and the TRIM and LEFT functions. However, using the CALL CATS, CALL CATT, and CALL CATX routines is faster than using TRIM and LEFT.

The following table shows statements that are equivalent to CALL CATS, CALL CATT, and CALL CATX. The variables X1 through X4 specify character variables, and SP specifies a separator, such as a blank or comma.

<table>
<thead>
<tr>
<th>CALL Routine</th>
<th>Equivalent Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL CATS(res, OF X1-X4);</td>
<td>res=TRIM(LEFT(X1))</td>
</tr>
<tr>
<td>CALL CATT(OF X1-X4);</td>
<td>X1=TRIM(X1)</td>
</tr>
<tr>
<td>CALL CATX(SP, OF X1-X4);</td>
<td>* X1=TRIM(LEFT(X1))</td>
</tr>
</tbody>
</table>

* If any of the arguments is blank, the results that are produced by CALL CATX differ slightly from the results that are produced by the concatenated code. In this case, CALL CATX omits the corresponding separator. For example, CALL CATX("+","X","","Z"," "); produces X+Z.

Examples
The following example shows how the CALL CATT routine concatenates strings.
data _null_;  
  length answer $ 36;  
  x='Athens is t  ';  
  y='he Olym  ';  
  z='pic site for 2004. ';  
  call catt(answer,x,y,z);  
  put answer;  
run;  

The following line is written to the SAS log:  

----+----1----+----2----+----3----+----4  
Athens is the Olympic site for 2004.

See Also

Functions and CALL Routines:  
“CAT Function” on page 408  
“CATS Function” on page 410  
“CATT Function” on page 412  
“CATX Function” on page 414  
“CALL CATS Routine” on page 337  
“CALL CATX Routine” on page 340

CALL CATX Routine

Concatenates character strings, removes leading and trailing blanks, and inserts separators

Category: Character

Syntax

CALL CATX(separator, result<, string-1 , ...>string-n);

Arguments

separator  
specifies a character string that is used as a separator between concatenated strings.

result  
specifies a SAS variable.

string  
specifies a SAS character string.

Details

The CALL CATX routine returns the result in the second argument,result. The routine appends the values of the arguments that follow to result. If the length of result is not large enough to contain the entire result, SAS
- writes a warning message to the log stating that the result was truncated
- writes a note to the log that shows the location of the function call and lists the argument that caused the truncation, except in SQL or in a WHERE clause
- sets _ERROR_ to 1 in the DATA step, except in a WHERE clause.

The CALL CATX routine removes leading and trailing blanks from numeric arguments after formatting the numeric value with the BEST format.

**Comparisons**

The results of the CALL CATS, CALL CATT, and CALL CATX routines are usually equivalent to statements that use the concatenation operator (||) and the TRIM and LEFT functions. However, using the CALL CATS, CALL CATT, and CALL CATX routines is faster than using TRIM and LEFT.

The following table shows statements that are equivalent to CALL CATS, CALL CATT, and CALL CATX. The variables X1 through X4 specify character variables, and SP specifies a separator, such as a blank or comma.

<table>
<thead>
<tr>
<th>CALL Routine</th>
<th>Equivalent Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL CATS(res, OF X1-X4);</td>
<td>res=TRIM(LEFT(X1))</td>
</tr>
<tr>
<td>CALL CATT(OF X1-X4);</td>
<td>X1=TRIM(X1)</td>
</tr>
<tr>
<td>CALL CATX(SP, OF X1-X4);</td>
<td>* X1=TRIM(LEFT(X1))</td>
</tr>
</tbody>
</table>

* If any of the arguments is blank, the results that are produced by CALL CATX differ slightly from the results that are produced by the concatenated code. In this case, CALL CATX omits the corresponding separator. For example, CALL CATX(“+”, “X”, “”, “Z”, “ ”); produces X+Z.

**Examples**

The following example shows how the CALL CATX routine concatenates strings.

```sas
data _null_
   length answer $ 50;
   separator='%%$%%';
   x='Athens is t ';
   y='he Olym ';
   z='pic site for 2004. ';
   answer=catx(separator,answer,x,y,z);
   put answer;
run;
```

The following line is written to the SAS log:

```
---+---1---2---3---4---5
Athens is t%%$%he Olym%%$%pic site for 2004.
```
See Also

Functions and CALL Routines:
  “CAT Function” on page 408
  “CATS Function” on page 410
  “CATT Function” on page 412
  “CATX Function” on page 414
  “CALL CATS Routine” on page 337
  “CALL CATT Routine” on page 338

CALL COMPCOST Routine

Sets the costs of operations for later use by the COMPGED function

Category: Character
Restriction: Use with the COMPGED function

Syntax

CALL COMPCOST(operation-1, value-1 <,operation-2, value-2 ...>);

Arguments

operation
  is a character constant, variable, or expression that specifies an operation that is performed by the COMPGED function.

value
  is a numeric constant, variable, or expression that specifies the cost of the operation that is indicated by the preceding argument.

Restriction: Must be an integer that ranges from -32767 to 32767, or a missing value

Details

Computing the Cost of Operations    Each argument that specifies an operation must have a value that is a character string. The character string corresponds to one of the terms that is used to denote an operation that the COMPGED function performs. See “Computing the Generalized Edit Distance” on page 448 to view a table of operations that the COMPGED function uses.

The character strings that specify operations can be in uppercase, lowercase, or mixed case. Blanks are ignored. Each character string must end with an equal sign (=). Valid values for operations, and the default cost of the operations are listed in the following table.
Functions and CALL Routines

## CALL COMPCOST Routine

### Operation Default Cost

<table>
<thead>
<tr>
<th>Operation</th>
<th>Default Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPEND=</td>
<td>very large</td>
</tr>
<tr>
<td>BLANK=</td>
<td>very large</td>
</tr>
<tr>
<td>DELETE=</td>
<td>100</td>
</tr>
<tr>
<td>DOUBLE=</td>
<td>very large</td>
</tr>
<tr>
<td>FDELETE=</td>
<td>equal to DELETE</td>
</tr>
<tr>
<td>FINSERT=</td>
<td>equal to INSERT</td>
</tr>
<tr>
<td>FREPLACE=</td>
<td>equal to REPLACE</td>
</tr>
<tr>
<td>INSERT=</td>
<td>100</td>
</tr>
<tr>
<td>MATCH=</td>
<td>0</td>
</tr>
<tr>
<td>PUNCTUATION=</td>
<td>very large</td>
</tr>
<tr>
<td>REPLACE=</td>
<td>100</td>
</tr>
<tr>
<td>SINGLE=</td>
<td>very large</td>
</tr>
<tr>
<td>SWAP=</td>
<td>very large</td>
</tr>
<tr>
<td>TRUNCATE=</td>
<td>very large</td>
</tr>
</tbody>
</table>

If an operation does not appear in the call to the COMPCOST routine, or if the operation appears and is followed by a missing value, then that operation is assigned a default cost. A “very large” cost indicates a cost that is sufficiently large that the COMPGED function will not use the corresponding operation.

After your program calls the COMPCOST routine, the costs that are specified remain in effect until your program calls the COMPCOST routine again, or until the step that contains the call to COMPCOST terminates.

### Abbreviating Character Strings

You can abbreviate character strings. That is, you can use the first one or more letters of a specific operation rather than use the entire term. You must, however, use as many letters as necessary to uniquely identify the term. For example, you can specify the INSERT= operation as “in=”, and the REPLACE= operation as “r=”. To specify the DELETE= or the DOUBLE= operation, you must use the first two letters because both DELETE= and DOUBLE= begin with “d”. The character string must always end with an equal sign.

### Examples

The following example calls the COMPCOST routine to compute the generalized edit distance for the operations that are specified.

```sas
options pageno=1 nodate linesize=80 pagesize=60;

data test;
  length String $8 Operation $40;
  if _n_ = 1 then call compcost('insert=',10,'DEL=',11,'r=',12);
  input String Operation;
  GED=compged(string, 'baboon');
  datalines;
  baboon match
  xbaboon insert
```

---

---
The following output shows the results.

### Output 4.4 Generalized Edit Distance Based on Operation

<table>
<thead>
<tr>
<th>Obs</th>
<th>String</th>
<th>Operation</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>baboon</td>
<td>match</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>xbaboon</td>
<td>insert</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>babon</td>
<td>delete</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>baXoon</td>
<td>replace</td>
<td>12</td>
</tr>
</tbody>
</table>

### See Also

Functions:

- “COMPGED Function” on page 447
- “COMPARE Function” on page 443
- “COMPLEV Function” on page 452

---

**CALL EXECUTE Routine**

Resolves an argument and issues the resolved value for execution

**Category:** Macro

**Syntax**

```
CALL EXECUTE(argument);
```

**Arguments**

`argument`

specifies a character expression or a constant that yields a macro invocation or a SAS statement. Argument can be:

- a character string, enclosed in quotation marks.
the name of a DATA step character variable. Do not enclose the name of the DATA step variable in quotation marks.

- a character expression that the DATA step resolves to a macro text expression or a SAS statement.

**Details**

If *argument* resolves to a macro invocation, the macro executes immediately and DATA step execution pauses while the macro executes. If *argument* resolves to a SAS statement or if execution of the macro generates SAS statements, the statement(s) execute after the end of the DATA step that contains the CALL EXECUTE routine. CALL EXECUTE is fully documented in *SAS Macro Language: Reference*.

---

### CALL LABEL Routine

**Assigns a variable label to a specified character variable**

**Category:** Variable Control

---

### Syntax

CALL LABEL(variable-1, variable-2);

### Arguments

- **variable-1**
  - specifies any SAS variable. If *variable-1* does not have a label, the variable name is assigned as the value of *variable-2*.

- **variable-2**
  - specifies any SAS character variable. Variable labels can be up to 256 characters long; therefore, the length of *variable-2* should be at least 256 characters to avoid truncating variable labels.

  **Note:** To conserve space, you should set the length of *variable-2* to the length of the label for *variable-1*, if it is known. △

### Details

The CALL LABEL routine assigns the label of the *variable-1* variable to the character variable *variable-2*.

### Examples

This example uses the CALL LABEL routine with array references to assign the labels of all variables in the data set OLD as values of the variable LAB in data set NEW:

```sas
data new;
  set old;
  /* lab is not in either array */
```

---

---
length lab $256;
   /* all character variables in old */
array abc{*} _character_;
   /* all numeric variables in old */
array def{*} _numeric_;
do i=1 to dim(abc);
   /* get label of character variable */
   call label(abc{i},lab);
   /* write label to an observation */
   output;
end;
do j=1 to dim(def);
   /* get label of numeric variable */
   call label(def{j},lab);
   /* write label to an observation */
   output;
end;
stop;
keep lab;
run;

See Also

Function:
   “V LABEL Function” on page 904

CALL LOGISTIC Routine

Returns the logistic value
Category: Mathematical

Syntax
CALL LOGISTIC(argument<, argument, ...>)

Arguments

argument
is numeric.
Restriction The CALL LOGISTIC routine only accepts variables as valid arguments. Do not use a constant or a SAS expression since the CALL routine is unable to update these arguments.

Details
The CALL LOGISTIC routine replaces each argument by the logistic value of that argument. For example \( x_j \) is replaced by
If any argument contains a missing value, then CALL LOGISTIC returns missing values for all the arguments.

**Examples**

The following SAS statements produce these results.

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=0.5;</td>
<td></td>
</tr>
<tr>
<td>y=-0.5;</td>
<td></td>
</tr>
<tr>
<td>call logistic(x,y);</td>
<td>x=0.6224593312 y=0.3775406688</td>
</tr>
<tr>
<td>put x= y=;</td>
<td></td>
</tr>
</tbody>
</table>
## Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>prod=’shoes’;</td>
<td></td>
</tr>
<tr>
<td>invty=7498;</td>
<td></td>
</tr>
<tr>
<td>sales=23759;</td>
<td></td>
</tr>
<tr>
<td>call missing(sales);</td>
<td></td>
</tr>
<tr>
<td>put prod= invty= sales=;</td>
<td>prod=shoes invty=7498 sales=.</td>
</tr>
<tr>
<td>prod=’shoes’;</td>
<td></td>
</tr>
<tr>
<td>invty=7498;</td>
<td></td>
</tr>
<tr>
<td>sales=23759;</td>
<td></td>
</tr>
<tr>
<td>call missing(prod,invty);</td>
<td></td>
</tr>
<tr>
<td>put prod= invty= sales=;</td>
<td>prod= invty=. sales=23759</td>
</tr>
<tr>
<td>prod=’shoes’;</td>
<td></td>
</tr>
<tr>
<td>invty=7498;</td>
<td></td>
</tr>
<tr>
<td>sales=23759;</td>
<td></td>
</tr>
<tr>
<td>call missing(of <em>all</em>);</td>
<td></td>
</tr>
<tr>
<td>put prod= invty= sales=;</td>
<td>prod= invty=. sales=.</td>
</tr>
</tbody>
</table>

## See Also

Function:

“MISSING Function” on page 647

“How to Set Variable Values to Missing in a Data Step” in *SAS Language Reference: Concepts*

## CALL MODULE Routine

Calls the external routine without any return code

**Category:** External Routines

### Syntax

`CALL MODULE(<cntl-string>,module-name<,argument-1, ..., argument-n>);`

### Arguments

**cntl-string**

is an optional control string whose first character must be an asterisk (*), followed by any combination of the following characters:

- `I` prints the hexadecimal representations of all arguments to the CALL MODULE routine. You can use this option to help diagnose
problems caused by incorrect arguments or attribute tables. If you specify the I option, the E option is implied.

E prints detailed error messages. Without the E option (or the I option, which supersedes it), the only error message that the CALL MODULE routine generates is “Invalid argument to function,” which is usually not enough information to determine the cause of the error. The E option is useful for a production environment, while the I option is preferable for a development or debugging environment.

H provides brief help information about the syntax of the CALL MODULE routine, the attribute file format, and suggested SAS formats and informats.

module-name is the name of the external module to use.

argument is one or more arguments to pass to the requested routine.

CAUTION: Be sure to use the correct arguments and attributes. If you use incorrect arguments or attributes, you can cause the SAS System, and possibly your operating system, to fail.

Details

The CALL MODULE routine executes a routine module-name that resides in an external library with the specified arguments.

CALL MODULE builds a parameter list using the information in the arguments and a routine description and argument attribute table that you define in a separate file. The attribute table is a sequential text file that contains descriptions of the routines that you can invoke with the CALL MODULE routine. The purpose of the table is to define how CALL MODULE should interpret its supplied arguments when it builds a parameter list to pass to the external routine. The attribute table should contain a description for each external routine that you intend to call, and descriptions of each argument associated with that routine.

Before you invoke CALL MODULE, you must define the fileref of SASCBTBL to point to the external file that contains the attribute table. You can name the file whatever you want when you create it. This way, you can use SAS variables and formats as arguments to CALL MODULE and ensure that these arguments are properly converted before being passed to the external routine. If you do not define this fileref, CALL MODULE calls the requested routine without altering the arguments.

CAUTION: Using the CALL MODULE routine without a defined attribute table can cause the SAS System to fail or force you to reset your computer. You need to use an attribute table for all external functions that you want to invoke.

Comparisons

The two CALL routines and four functions share identical syntax:

□ The MODULEN and MODULEC functions return a number and a character, respectively, while the routine CALL MODULE does not return a value.

□ The CALL MODULEI routine and the functions MODULEIC and MODULEIN permit vector and matrix arguments. Their return values are scalar. You can
invoke CALL MODULEI, MODULEIC, and MODULEIN only from the IML procedure.

**Examples**

**Example 1: Using the CALL MODULE Routine**

This example calls the *xyz* routine. Use the following attribute table:

```sas
routine xyz minarg=2 maxarg=2;
arg 1 input num byvalue format=ib4.;
arg 2 output char format=$char10.;
```

The following is the sample SAS code that calls the *xyz* function:

```sas
data _null_;  
call module(‘xyz’,1,x);  
run;
```

**Example 2: Using the MODULEIN Function in the IML Procedure**

This example invokes the *changi* routine from the TRYMOD.DLL module on a Windows platform. Use the following attribute table:

```sas
routine changi module=trymod returns=long;
arg 1 input num format=ib4. byvalue;
arg 2 update num format=ib4.;
```

The following PROC IML code calls the *changi* function:

```sas
proc iml;  
x1=J(4,5,0);  
do i=1 to 4;  
  do j=1 to 5;  
    x1[i,j]=i*10+j+3;  
  end;  
end;  
y1=x1;  
x2=x1;  
y2=y1;  
rc=modulein(‘*i’,’changi’,6,x2);
```

**Example 3: Using the MODULEN Function**

This example calls the *Beep* routine, which is part of the Win32 API in the KERNEL32 Dynamic Link Library on a Windows platform. Use the following attribute table:

```sas
routine Beep  
  minarg=2  
  maxarg=2  
  stackpop=called  
  callseq=byvalue  
  module=kernel32;  
arg 1 num format=pib4.;  
arg 2 num format=pib4.;
```

Assume that you name the attribute table file ‘myatttbl.dat’. The following is the sample SAS code that calls the *Beep* function:

```sas
filename sasctbtl ‘myatttbl.dat’;  
data _null_;
rc=modulen("*e","Beep",1380,1000);
run;

The previous code causes the computer speaker to beep.

See Also

CALL Routine:
“CALL MODULEI Routine” on page 351

Functions:
“MODULEC Function” on page 651
“MODULEIC Function” on page 651
“MODULEIN Function” on page 652
“MODULEN Function” on page 652

CALL MODULEI Routine

Calls the external routine without any return code (in IML environment only)

Category: External Routines
Restriction: CALL MODULEI can only be invoked from within the IML procedure
See: “CALL MODULE Routine” on page 348

Syntax

CALL MODULEI(<cntl-string,>module-name<,argument-1, ..., argument-n>);

Details

For details on CALL MODULEI, see “CALL MODULE Routine” on page 348.

See Also

CALL Routine:
“CALL MODULE Routine” on page 348

Functions:
“MODULEC Function” on page 651
“MODULEIC Function” on page 651
“MODULEIN Function” on page 652
“MODULEN Function” on page 652
CALL POKE Routine

Writes a value directly into memory on a 32–bit platform

Category: Special
Restriction: Use on 32–bit platforms only.

Syntax

CALL POKE(source, pointer<, length>);

Arguments

source
specifies a SAS expression that contains a value to write into memory.

pointer
specifies a numeric SAS expression that contains the virtual address of the data that the CALL POKE routine alters.

length
specifies a numeric SAS expression that contains the number of bytes to write from the source to the address that is indicated by pointer. If you omit length, the action that the CALL POKE routine takes depends on whether source is a character value or a numeric value:

- If source is a character value, the CALL POKE routine copies the entire value of source to the specified memory location.
- If source is a numeric value, the CALL POKE routine converts source into a long integer and writes into memory the number of bytes that constitute a pointer.

Operating Environment Information: Under z/OS, pointers are 3 or 4 bytes long, depending on the situation.

Details

CAUTION:
The CALL POKE routine is intended only for experienced programmers in specific cases. If you plan to use this routine, use extreme care both in your programming and in your typing. Writing directly into memory can cause devastating problems. This routine bypasses the normal safeguards that prevent you from destroying a vital element in your SAS session or in another piece of software that is active at the time.

If you do not have access to the memory location that you specify, the CALL POKE routine returns an "Invalid argument" error.

You cannot use the CALL POKE routine on 64-bit platforms. If you attempt to use it, SAS writes a message to the log stating that this restriction applies. If you have legacy applications that use CALL POKE, change the applications and use CALL POKELONG instead. You can use CALL POKELONG on both 32–bit and 64–bit platforms.

See Also

Functions:
CALL POKELONG Routine

Writes a value directly into memory on 32-bit and 64-bit platforms

Category:  Special

Syntax

CALL POKELONG(source, pointer<, length>)

Arguments

source
specifies a character string that contains a value to write into memory.

pointer
specifies a character string that contains the virtual address of the data that the CALL POKELONG routine alters.

length
specifies a numeric SAS expression that contains the number of bytes to write from the source to the address that is indicated by the pointer. If you omit length, the CALL POKELONG routine copies the entire value of source to the specified memory location.

Details

CAUTION:
The CALL POKELONG routine is intended only for experienced programmers in specific cases. If you plan to use this routine, use extreme care both in your programming and in your typing. Writing directly into memory can cause devastating problems. It bypasses the normal safeguards that prevent you from destroying a vital element in your SAS session or in another piece of software that is active at the time.

If you do not have access to the memory location that you specify, the CALL POKELONG routine returns an "Invalid argument" error.

CALL PRXCHANGE Routine

Performs a pattern-matching replacement

Category:  Character String Matching

Restriction:  Use with the PRXPARSE function.
Syntax

CALL PRXCHANGE (regular-expression-id, times, old-string <, new-string <, result-length <, truncation-value <, number-of-changes>>);

Arguments

regular-expression-id
specifies a numeric pattern identifier that is returned from the PRXPARSE function.

times
is a numeric value that specifies the number of times to search for a match and replace a matching pattern.
Tip: If the value of times is -1, then all matching patterns are replaced.

old-string
specifies the character expression on which to perform a search and replace.
Tip: All changes are made to old-string if you do not use the new-string argument.

new-string
specifies a character variable in which to place the results of the change to old-string.
Tip: If you use the new-string argument in the call to the PRXCHANGE routine, then old-string is not modified.

result-length
is a numeric variable that specifies the number of characters that are copied into the result.
Tip: Trailing blanks in the value of old-string are not copied to new-string, and are therefore not included as part of the length in result-length.

truncation-value
is either 0 or 1, depending on the result of the change operation:
0 if the entire replacement result is not longer than the length of new-string.
1 if the entire replacement result is longer than the length of new-string.

number-of-changes
is a numeric variable that specifies the total number of replacements that were made. If the result is truncated when it is placed into new-string, the value of number-of-changes is not changed.

Details

The CALL PRXCHANGE routine matches and replaces a pattern. If the value of times is -1, the replacement is performed as many times as possible.
For more information about pattern matching, see “Pattern Matching Using SAS Regular Expressions (RX) and Perl Regular Expressions (PRX)” on page 260.

Comparisons

The CALL PRXCHANGE routine is similar to the PRXCHANGE function except that the CALL routine returns the value of the pattern matching replacement as one of its parameters instead of as a return argument.
The Perl regular expression (PRX) functions and CALL routines work together to manipulate strings that match patterns. To see a list and short description of these functions and CALL routines, see the Character String Matching category in “Functions and CALL Routines by Category” on page 270.

Examples

The following example replaces all occurrences of cat, rat, or bat with the value TREE.

```sas
data _null_; /* Use a pattern to replace all occurrences of cat, */ /* rat, or bat with the value TREE. */ length text $ 46; RegularExpressionId = prxparse('s/[crb]at/tree/'); text = 'The woods have a bat, cat, bat, and a rat!'; /* Use CALL PRXCHANGE to perform the search and replace. */ /* Because the argument times has a value of -1, the */ /* replacement is performed as many times as possible. */ call prxchange(RegularExpressionId, -1, text); put text; run;
```

SAS writes the following line to the log:

The woods have a tree, tree, tree, and a tree!

See Also

Functions and CALL routines:

- “CALL PRXDEBUG Routine” on page 355
- “CALL PRXFREE Routine” on page 358
- “CALL PRXNEXT Routine” on page 359
- “CALL PRXPOSN Routine” on page 361
- “CALL PRXSUBSTR Routine” on page 363
- “PRXCHANGE Function” on page 739
- “PRXPAREN Function” on page 747
- “PRXMATCH Function” on page 743
- “PRXPARSE Function” on page 749
- “PRXPOSN Function” on page 751

CALL PRXDEBUG Routine

Enables Perl regular expressions in a DATA step to send debug output to the SAS log

Category: Character String Matching

Restriction: Use with the CALL PRXCHANGE, CALL PRXFREE, CALL PRXNEXT, CALL PRXPOSN, CALL PRXSUBSTR, PRXPARSE, PRXPAREN, and PRXMATCH functions and CALL routines.
**Syntax**

CALL PRXDEBUG (on-off);

**Arguments**

*on-off*

specifies a numeric value. If the value of *on-off* is positive and non-zero, then debugging is turned on. If the value of *on-off* is zero, then debugging is turned off.

**Details**

The CALL PRXDEBUG routine provides information about how a Perl regular expression is compiled, and about which steps are taken when a pattern is matched to a character value.

You can turn debugging on and off multiple times in your program if you want to see debugging output for particular Perl regular expression function calls.

For more information about pattern matching, see “Pattern Matching Using SAS Regular Expressions (RX) and Perl Regular Expressions (PRX)” on page 260.

**Comparisons**

The Perl regular expression (PRX) functions and CALL routines work together to manipulate strings that match patterns. To see a list and short description of these functions and CALL routines, see the Character String Matching category in “Functions and CALL Routines by Category” on page 270.

**Examples**

The following example produces debugging output.

```sas
data _null_;  
    /* Turn the debugging option on. */
    call prxdebug(1);  
    putlog 'PRXPARSE: ';  
    re = prxparse('/[bc]d(ef*g)+h[ij]k$/');  
    putlog 'PRXMATCH: ';  
    pos = prxmatch(re, 'abcdefg_gh_');  
    /* Turn the debugging option off. */
    call prxdebug(0);  
run;  
```

The following lines are written to the SAS log.
Output 4.5  SAS Log Results from CALL PRXDEBUG

<table>
<thead>
<tr>
<th>PRXPARSE:</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiling REx '{bc}d{ef*g}+h[ij]k$'</td>
<td></td>
</tr>
<tr>
<td>size 41 first at 1</td>
<td>2</td>
</tr>
<tr>
<td>rarest char g at 0</td>
<td></td>
</tr>
<tr>
<td>rarest char d at 0</td>
<td>3</td>
</tr>
<tr>
<td>1: ANYOF[b]{10}</td>
<td></td>
</tr>
<tr>
<td>10: EXACT &lt;d&gt;</td>
<td>12</td>
</tr>
<tr>
<td>12: CURLYX[0] {1,32767}</td>
<td>26</td>
</tr>
<tr>
<td>14: OPEN1(16)</td>
<td></td>
</tr>
<tr>
<td>16: EXACT &lt;e&gt;{18}</td>
<td></td>
</tr>
<tr>
<td>18: STAR(21)</td>
<td></td>
</tr>
<tr>
<td>19: EXACT &lt;f&gt;{0}</td>
<td></td>
</tr>
<tr>
<td>21: EXACT &lt;g&gt;{23}</td>
<td></td>
</tr>
<tr>
<td>23: CLOSE1(25)</td>
<td></td>
</tr>
<tr>
<td>25: WHILEM[1/1]0</td>
<td></td>
</tr>
<tr>
<td>26: NOTHING(27)</td>
<td></td>
</tr>
<tr>
<td>27: EXACT &lt;h&gt;{29}</td>
<td></td>
</tr>
<tr>
<td>29: ANYOF[ij]{38}</td>
<td></td>
</tr>
<tr>
<td>38: EXACT &lt;k&gt;{40}</td>
<td></td>
</tr>
<tr>
<td>40: EOL{41}</td>
<td></td>
</tr>
<tr>
<td>41: END(0)</td>
<td></td>
</tr>
<tr>
<td>anchored ‘de’ at 1 floating ‘gh’ at 3..2147483647 (checking floating)</td>
<td>4</td>
</tr>
<tr>
<td>stclss ‘ANYOF[b]’ minlen 7</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PRXMATCH:</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guessing start of match, REx '{bc}d{ef*g}+h[ij]k$' against ‘abcdefg_gh_’...</td>
<td></td>
</tr>
<tr>
<td>Did not find floating substr ‘gh’...</td>
<td></td>
</tr>
<tr>
<td>Match rejected by optimizer</td>
<td></td>
</tr>
</tbody>
</table>

The following items correspond to the lines that are numbered in the SAS log that is shown above.

1. This line shows the precompiled form of the Perl regular expression.
2. Size specifies a value in arbitrary units of the compiled form of the Perl regular expression. 41 is the label ID of the first node that performs a match.
3. This line begins a list of program nodes in compiled form for regular expressions.
4. These two lines provide optimizer information. In the example above, the optimizer found that the match should contain the substring de at offset 1, and the substring gh at an offset between 3 and infinity. To rule out a pattern match quickly, Perl checks substring gh before it checks substring de.

The optimizer might use the information that the match begins at the first ID (2), with a character class (4), and cannot be shorter than seven characters (6).

See Also

Functions and CALL routines:

“CALL PRXCHANGE Routine” on page 353
“CALL PRXFREE Routine” on page 358
“CALL PRXNEXT Routine” on page 359
“CALL PRXPOSN Routine” on page 361
“CALL PRXSUBSTR Routine” on page 363
“CALL PRXCHANGE Routine” on page 353
CALL PRXFREE Routine

Frees unneeded memory that was allocated for a Perl regular expression

**Category:**  Character String Matching
**Restriction:**  Use with the PRXPARSE function

**Syntax**

CALL PRXFREE (regular-expression-id);

**Arguments**

**regular-expression-id**

specifies a numeric identification number that is returned by the PRXPARSE function. regular-expression-id is set to missing if the call to the PRXFREE routine occurs without error.

**Details**

The CALL PRXFREE routine frees unneeded resources that were allocated for a Perl regular expression.

For more information about pattern matching, see “Pattern Matching Using SAS Regular Expressions (RX) and Perl Regular Expressions (PRX)” on page 260.

**Comparisons**

The Perl regular expression (PRX) functions and CALL routines work together to manipulate strings that match patterns. To see a list and short description of these functions and CALL routines, see the Character String Matching category in “Functions and CALL Routines by Category” on page 270.

**See Also**

Functions and CALL routines:

“CALL PRXCHANGE Routine” on page 353
“CALL PRXDEBUG Routine” on page 355
“CALL PRXNEXT Routine” on page 359
“CALL PRXPOSN Routine” on page 361
CALL PRXNEXT Routine

Returns the position and length of a substring that matches a pattern and iterates over multiple matches within one string

Category: Character String Matching
Restriction: Use with the PRXPARSE function

Syntax
CALL PRXNEXT (regular-expression-id, start, stop, source, position, length);

Arguments

regular-expression-id
specifies a numeric identification number that is returned by the PRXPARSE function.

start
is a numeric value that specifies the position at which to start the pattern matching in source. If the match is successful, CALL PRXNEXT returns a value of position + MAX(1, length). If the match is not successful, the value of start is not changed.

stop
is a numeric value that specifies the last character to use in source. If stop is -1, then the last character is the last non-blank character in source.

source
specifies the character expression that you want to search.

position
specifies the numeric position in source at which the pattern begins. If no match is found, CALL PRXNEXT returns zero.

length
specifies a numeric value that is the length of the string that is matched by the pattern. If no match is found, CALL PRXNEXT returns zero.

Details
The CALL PRXNEXT routine searches the variable source with a pattern. It returns the position and length of a pattern match that is located between the start and the
stop positions in source. Because the value of the start parameter is updated to be the position of the next character that follows a match, CALL PRXNEXT enables you to search a string for a pattern multiple times in succession.

For more information about pattern matching, see “Pattern Matching Using SAS Regular Expressions (RX) and Perl Regular Expressions (PRX)” on page 260.

Comparisons

The Perl regular expression (PRX) functions and CALL routines work together to manipulate strings that match patterns. To see a list and short description of these functions and CALL routines, see the Character String Matching category in “Functions and CALL Routines by Category” on page 270.

Examples

The following example finds all instances of cat, rat, or bat in a text string.

```sas
data _null_
  expressionid = prxparse('/[crb]at/');
  text = 'The woods have a bat, cat, and a rat!';
  start = 1;
  stop = length(text);
  /* Use PRXNEXT to find the first instance of the pattern, */
  /* then use DO WHILE to find all further instances. */
  /* PRXNEXT changes the start parameter so that searching */
  /* begins again after the last match. */
  call prxnext(expressionid, start, stop, text, position, length);
  do while (position > 0);
    found = substr(text, position, length);
    put found= position= length=;
    call prxnext(expressionid, start, stop, text, position, length);
  end;
run;
```

The following lines are written to the SAS log:

```
found=bat position=18 length=3
found=cat position=23 length=3
found=rat position=34 length=3
```

See Also

Functions and CALL routines:

“CALL PRXCHANGE Routine” on page 353
“CALL PRXDEBUG Routine” on page 355
“CALL PRXFREE Routine” on page 358
“CALL PRXPOSN Routine” on page 361
“CALL PRXSUBSTR Routine” on page 363
“CALL PRXCHANGE Routine” on page 353
“PRXCHANGE Function” on page 739
“PRXPAREN Function” on page 747
CALL PRXPOSN Routine

Returns the start position and length for a capture buffer

Category: Character String Matching
Restriction: Use with the PRXPARSE function

Syntax

CALL PRXPOSN (regular-expression-id, capture-buffer, start <, length>);

Arguments

**regular-expression-id**
specifies a numeric pattern identifier that is returned by the PRXPARSE function.

**capture-buffer**
is a numeric value that identifies the capture buffer from which to retrieve the start position and length:
- If the value of capture-buffer is zero, CALL PRXPOSN returns the start position and length of the entire match.
- If the value of capture-buffer is between 1 and the number of open parentheses, CALL PRXPOSN returns the start position and length for that capture buffer.
- If the value of capture-buffer is greater than the number of open parentheses, CALL PRXPOSN returns missing values for the start position and length.

**start**
is a numeric value that specifies the position at which the capture buffer is found:
- If the value of capture-buffer is not found, CALL PRXPOSN returns a zero value for the start position.
- If the value of capture-buffer is greater than the number of open parentheses in the pattern, CALL PRXPOSN returns a missing value for the start position.

**length**
is a numeric value that specifies the pattern length of the previous pattern match:
- If the pattern match is not found, CALL PRXPOSN returns a zero value for the length.
- If the value of capture-buffer is greater than the number of open parentheses in the pattern, CALL PRXPOSN returns a missing value for length.

Details

The CALL PRXPOSN routine uses the results of PRXMATCH, PRXSUBSTR, PRXCHANGE, or PRXNEXT to return a capture buffer. A match must be found by one of these functions for the CALL PRXPOSN routine to return meaningful information.
A capture buffer is part of a match, enclosed in parentheses, that is specified in a regular expression. CALL PRXPOSN does not return the text for the capture buffer directly. It requires a call to the SUBSTR function to return the text.

For more information about pattern matching, see “Pattern Matching Using SAS Regular Expressions (RX) and Perl Regular Expressions (PRX)” on page 260.

**Comparisons**

The CALL PRXPOSN routine is similar to the PRXPOSN function, except that CALL PRXPOSN returns the position and length of the capture buffer rather than the capture buffer itself.

The Perl regular expression (PRX) functions and CALL routines work together to manipulate strings that match patterns. To see a list and short description of these functions and CALL routines, see the Character String Matching category in “Functions and CALL Routines by Category” on page 270.

**Examples**

**Example 1: Finding Submatches within a Match**  
The following example searches a regular expression and calls the PRXPOSN routine to find the position and length of three submatches.

```sas
data _null_
  patternID = prxparse('/\d\d:\d\d(am|pm)/');
  text = 'The time is 09:56am.';
  if prxmatch(patternID, text) then do;
    call prxposn(patternID, 1, position, length);
    hour = substr(text, position, length);
    call prxposn(patternID, 2, position, length);
    minute = substr(text, position, length);
    call prxposn(patternID, 3, position, length);
    ampm = substr(text, position, length);
    put hour= minute= ampm=;
    put text=;
  end;
run;
```

SAS writes the following lines to the log:

```
hour=09 minute=56 ampm=am
text=The time is 09:56am.
```

**Example 2: Parsing Time Data**  
The following example parses time data and writes the results to the SAS log.

```sas
data _null_
  if _N_ = 1 then do;
    retain patternID;
    pattern = "/(\d+):(\d\d)\d\d/";
    patternID = prxparse(pattern);
  end;
```
array match[3] $ 8;
input minsec $80.;
position = prxmatch(patternID, minsec);
if position ^= 0 then
do;
   do i = 1 to prxparen(patternID);
      call prxposn(patternID, i, start, length);
      if start ^= 0 then
         match[i] = substr(minsec, start, length);
   end;
   if "missing(match[3])" then
      put ", " match[3] "milliseconds";
end;
datalines;
14:56.456
45:32
;
SAS writes the following lines to the log:
14 minutes, 56 seconds, 456 milliseconds
45 minutes, 32 seconds

See Also

Functions and CALL routines:
   “CALL PRXCHANGE Routine” on page 353
   “CALL PRXDEBUG Routine” on page 355
   “CALL PRXFREE Routine” on page 358
   “CALL PRXNEXT Routine” on page 359
   “CALL PRXPARSE Routine” on page 359
   “CALL PRXSUBSTR Routine” on page 363
   “CALL PRXCHANGE Routine” on page 353
   “PRXCHANGE Function” on page 739
   “PRXPAREN Function” on page 747
   “PRXMATCH Function” on page 743
   “PRXPARSE Function” on page 749
   “PRXPOSN Function” on page 751

CALL PRXSUBSTR Routine

Returns the position and length of a substring that matches a pattern

Category: Character String Matching

Restriction: Use with the PRXPARSE function
Syntax

CALL PRXSUBSTR (regular-expression-id, source, position <, length>);

Arguments

regular-expression-id
  specifies a numeric identification number that is returned by the PRXPARSE
  function.

source
  specifies the character expression that you want to search.

position
  is a numeric value that specifies the position in source where the pattern begins. If
  no match is found, CALL PRXSUBSTR returns zero.

length
  specifies a numeric value that is the length of the substring that is matched by the
  pattern. If no match is found, CALL PRXSUBSTR returns zero.

Details

The CALL PRXSUBSTR routine searches the variable source with the pattern from
PRXPARSE, returns the position of the start of the string, and optionally returns the
length of the string that is matched. By default, when a pattern matches more than one
character that begins at a specific position, CALL PRXSUBSTR selects the longest
match.

For more information about pattern matching, see “Pattern Matching Using SAS
Regular Expressions (RX) and Perl Regular Expressions (PRX)” on page 260.

Comparisons

CALL PRXSUBSTR performs the same matching as PRXMATCH, but CALL
PRXSUBSTR additionally enables you to use the length argument to receive more
information about the match.

The Perl regular expression (PRX) functions and CALL routines work together to
manipulate strings that match patterns. To see a list and short description of these
functions and CALL routines, see the Character String Matching category in “Functions
and CALL Routines by Category” on page 270.

Examples

Example 1: Finding the Position and Length of a Substring
The following example searches a string for a substring, and returns its position and length in the string.

data _null_
  /* Use PRXPARSE to compile the Perl regular expression. */
  patternID = prxparse('/world/');
  /* Use PRXSUBSTR to find the position and length of the string. */
  call prxsubstr(patternID, 'Hello world!', position, length);
  put position= length=;
run;
The following line is written to the SAS log:

```
position=7 length=5
```

**Example 2: Finding a Match in a Substring** The following example searches for addresses that contain avenue, drive, or road, and extracts the text that was found.

```sas
data _null_;  
if _N_ = 1 then  
do;  
  retain ExpressionID;  
  /* The i option specifies a case insensitive search. */  
pattern = "/ave|avenue|dr|drive|rd|road/i";  
  ExpressionID = prxparse(pattern);  
end;  

input street $80.;  
call prxsubstr(ExpressionID, street, position, length);  
if position ^= 0 then  
do;  
  match = substr(street, position, length);  
  put match:$QUOTE. "found in " street:$QUOTE.;  
end;  
datalines;  
153 First Street  
6789 64th Ave  
4 Moritz Road  
7493 Wilkes Place  
;
run;
```

The following lines are written to the SAS log:

```
"Ave" found in "6789 64th Ave"
"Road" found in "4 Moritz Road"
```

**See Also**

Functions and CALL routines:

- “CALL PRXCHANGE Routine” on page 353
- “CALL PRXDEBUG Routine” on page 355
- “CALL PRXFREE Routine” on page 358
- “CALL PRXNEXT Routine” on page 359
- “CALL PRXPOSN Routine” on page 361
- “PRXCHANGE Function” on page 739
- “PRXPAREN Function” on page 747
- “PRXMATCH Function” on page 743
- “PRXPARSE Function” on page 749
- “PRXPOSN Function” on page 751
CALL RANBIN Routine

Returns a random variate from a binomial distribution

Category: Random Number

Syntax

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

Arguments

seed

is the seed value. A new value for seed is returned each time CALL RANBIN is executed.

Range: seed < 2^{31} - 1

Note: If seed ≤ 0, the time of day is used to initialize the seed stream.

See: “Seed Values” on page 257 for more information about seed values

n

is an integer number of independent Bernoulli trials.

Range: n > 0

p

is a numeric probability of success parameter.

Range: 0 < p < 1

x

is a numeric SAS variable. A new value for the random variate x is returned each time CALL RANBIN is executed.

Details

The CALL RANBIN routine updates seed and returns a variate x that is generated from a binomial distribution with mean np and variance np(1−p). If n ≤ 50, np ≤ 5, or n(1−p) ≤ 5, SAS uses an inverse transform method applied to a RANUNI uniform variate. If n > 50, np > 5, and n(1−p) > 5, SAS uses the normal approximation to the binomial distribution. In that case, the Box-Muller transformation of RANUNI uniform variates is used.

By adjusting the seeds, you can force streams of variates to agree or disagree for some or all of the observations in the same, or in subsequent, DATA steps.

Comparisons

The CALL RANBIN routine gives greater control of the seed and random number streams than does the RANBIN function.

Examples

This example uses the CALL RANBIN routine:
options nodate pageno=1 linesize=80 pagesize=60;

data case;
    retain Seed_1 Seed_2 Seed_3 45;
    n=2000;
    p=.2;
    do i=1 to 10;
        call ranbin(Seed_1,n,p,X1);
        call ranbin(Seed_2,n,p,X2);
        X3=ranbin(Seed_3,n,p);
        if i=5 then
            do;
                Seed_2=18;
                Seed_3=18;
            end;
        output;
    end;
run;

proc print;
    id i;
    var Seed_1-Seed_3 X1-X3;
run;

The following output shows the results:

Output 4.6  The RANBIN Example

<table>
<thead>
<tr>
<th>i</th>
<th>Seed_1</th>
<th>Seed_2</th>
<th>Seed_3</th>
<th>X1</th>
<th>X2</th>
<th>X3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1404437564</td>
<td>1404437564</td>
<td>45</td>
<td>385</td>
<td>385</td>
<td>385</td>
</tr>
<tr>
<td>2</td>
<td>1445125588</td>
<td>1445125588</td>
<td>45</td>
<td>399</td>
<td>399</td>
<td>399</td>
</tr>
<tr>
<td>3</td>
<td>1326029789</td>
<td>1326029789</td>
<td>45</td>
<td>384</td>
<td>384</td>
<td>384</td>
</tr>
<tr>
<td>4</td>
<td>1988843719</td>
<td>1988843719</td>
<td>45</td>
<td>421</td>
<td>421</td>
<td>421</td>
</tr>
<tr>
<td>5</td>
<td>2137808851</td>
<td>18</td>
<td>18</td>
<td>430</td>
<td>430</td>
<td>430</td>
</tr>
<tr>
<td>6</td>
<td>1233028129</td>
<td>991271755</td>
<td>18</td>
<td>392</td>
<td>374</td>
<td>392</td>
</tr>
<tr>
<td>7</td>
<td>50049159</td>
<td>1437043694</td>
<td>18</td>
<td>424</td>
<td>384</td>
<td>424</td>
</tr>
<tr>
<td>8</td>
<td>802575599</td>
<td>959908645</td>
<td>18</td>
<td>371</td>
<td>383</td>
<td>371</td>
</tr>
<tr>
<td>9</td>
<td>100573943</td>
<td>1225034217</td>
<td>18</td>
<td>428</td>
<td>388</td>
<td>428</td>
</tr>
<tr>
<td>10</td>
<td>414117170</td>
<td>425626811</td>
<td>18</td>
<td>402</td>
<td>403</td>
<td>402</td>
</tr>
</tbody>
</table>

Changing Seed_2 for the CALL RANBIN statement, when I=5, forces the stream of the variates for X2 to deviate from the stream of the variates for X1. Changing Seed_3 on the RANBIN function, however, has no effect.

See Also

Function:
    “RANBIN Function” on page 764
CALL RANCAU Routine

Returns a random variate from a Cauchy distribution

Category: Random Number

Syntax

CALL RANCAU(seed,x);

Arguments

seed
is the seed value. A new value for seed is returned each time CALL RANCAU is executed.

Range: seed < 2^{31} - 1

Note: If seed ≤ 0, the time of day is used to initialize the seed stream.

See: “Seed Values” on page 257 for more information about seed values

x
is a numeric SAS variable. A new value for the random variate x is returned each time CALL RANCAU is executed.

Details

The CALL RANCAU routine updates seed and returns a variate x that is generated from a Cauchy distribution that has a location parameter of 0 and scale parameter of 1. By adjusting the seeds, you can force streams of variates to agree or disagree for some or all of the observations in the same, or in subsequent, DATA steps.

An acceptance-rejection procedure applied to RANUNI uniform variates is used. If u and v are independent uniform (-1/2, 1/2) variables and u^2 + v^2 ≤ 1/4, then u/v is a Cauchy variate.

Comparisons

The CALL RANCAU routine gives greater control of the seed and random number streams than does the RANCAU function.

Examples

This example uses the CALL RANCAU routine:

options nodate pageno=1 linesize=80 pagesize=60;

data case;
   retain Seed_1 Seed_2 Seed_3 45;
   do i=1 to 10;
      call rancau(Seed_1,X1);
      call rancau(Seed_2,X2);
X3=rancau(Seed_3);
if i=5 then
do;
   Seed_2=18;
   Seed_3=18;
end;
end;
output;
run;

proc print;
  id i;
  var Seed_1-Seed_3 X1-X3;
run;

The following output shows the results:

### Output 4.7  The RANCAU Example

<table>
<thead>
<tr>
<th>i</th>
<th>Seed_1</th>
<th>Seed_2</th>
<th>Seed_3</th>
<th>X1</th>
<th>X2</th>
<th>X3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1404437564</td>
<td>1404437564</td>
<td>45</td>
<td>-1.14736</td>
<td>-1.14736</td>
<td>-1.14736</td>
</tr>
<tr>
<td>2</td>
<td>1326029789</td>
<td>1326029789</td>
<td>45</td>
<td>-0.23735</td>
<td>-0.23735</td>
<td>-0.23735</td>
</tr>
<tr>
<td>3</td>
<td>1988843719</td>
<td>1988843719</td>
<td>45</td>
<td>-0.15474</td>
<td>-0.15474</td>
<td>-0.15474</td>
</tr>
<tr>
<td>4</td>
<td>1233028129</td>
<td>1233028129</td>
<td>45</td>
<td>4.97935</td>
<td>4.97935</td>
<td>4.97935</td>
</tr>
<tr>
<td>5</td>
<td>50049159</td>
<td>18</td>
<td>18</td>
<td>0.20402</td>
<td>0.20402</td>
<td>0.20402</td>
</tr>
<tr>
<td>6</td>
<td>802575599</td>
<td>991271755</td>
<td>18</td>
<td>3.43645</td>
<td>4.44427</td>
<td>3.43645</td>
</tr>
<tr>
<td>7</td>
<td>1233458739</td>
<td>1437043694</td>
<td>18</td>
<td>6.32808</td>
<td>-1.79200</td>
<td>6.32808</td>
</tr>
<tr>
<td>8</td>
<td>52428589</td>
<td>959908645</td>
<td>18</td>
<td>0.18815</td>
<td>-1.67610</td>
<td>0.18815</td>
</tr>
<tr>
<td>9</td>
<td>1216356463</td>
<td>1225034217</td>
<td>18</td>
<td>3.80689</td>
<td>3.88391</td>
<td>0.80689</td>
</tr>
<tr>
<td>10</td>
<td>171188541</td>
<td>425626811</td>
<td>18</td>
<td>0.92971</td>
<td>-1.31309</td>
<td>0.92971</td>
</tr>
</tbody>
</table>

Changing Seed_2 for the CALL RANCAU statement, when I=5, forces the stream of the variates for X2 to deviate from the stream of the variates for X1. Changing Seed_3 on the RANCAU function, however, has no effect.

### See Also

Function:
“RANCAU Function” on page 765

---

### CALL RANEXP Routine

Returns a random variate from an exponential distribution

**Category:** Random Number

**Syntax**

CALL RANEXP(seed,x);
Arguments

\textit{seed}

is the seed value. A new value for \textit{seed} is returned each time CALL RANEXP is executed.

\textbf{Range:} \quad seed < 2^{31} - 1

\textbf{Note:} If seed \leq 0, the time of day is used to initialize the seed stream.

\textbf{See:} “Seed Values” on page 257 for more information about seed values

\textit{x}

is a numeric variable. A new value for the random variate \textit{x} is returned each time CALL RANEXP is executed.

Details

The CALL RANEXP routine updates \textit{seed} and returns a variate \textit{x} that is generated from an exponential distribution that has a parameter of 1.

By adjusting the seeds, you can force streams of variates to agree or disagree for some or all of the observations in the same, or in subsequent, DATA steps.

The CALL RANEXP routine uses an inverse transform method applied to a RANUNI uniform variate.

Comparisons

The CALL RANEXP routine gives greater control of the seed and random number streams than does the RANEXP function.

Examples

This example uses the CALL RANEXP routine:

\begin{verbatim}
options nodate pageno=1 linesize=80 pagesize=60;

data case;
  retain Seed_1 Seed_2 Seed_3 45;
  do i=1 to 10;
    call ranexp(Seed_1,X1);
    call ranexp(Seed_2,X2);
    X3=ranexp(Seed_3);
    if i=5 then
      do;
        seed_2=18;
        seed_3=18;
      end;
    output;
  end;
run;

proc print;
  id i;
  var Seed_1-Seed_3 X1-X3;
run;
\end{verbatim}
The following output shows the results:

**Output 4.8  The RANEXP Example**

<table>
<thead>
<tr>
<th>i</th>
<th>Seed_1</th>
<th>Seed_2</th>
<th>Seed_3</th>
<th>X1</th>
<th>X2</th>
<th>X3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>694315054</td>
<td>694315054</td>
<td>45</td>
<td>1.12913</td>
<td>1.12913</td>
<td>1.12913</td>
</tr>
<tr>
<td>2</td>
<td>1404437564</td>
<td>1404437564</td>
<td>45</td>
<td>0.42466</td>
<td>0.42466</td>
<td>0.42466</td>
</tr>
<tr>
<td>3</td>
<td>2130505156</td>
<td>2130505156</td>
<td>45</td>
<td>0.00794</td>
<td>0.00794</td>
<td>0.00794</td>
</tr>
<tr>
<td>4</td>
<td>1445125588</td>
<td>1445125588</td>
<td>45</td>
<td>0.39610</td>
<td>0.39610</td>
<td>0.39610</td>
</tr>
<tr>
<td>5</td>
<td>1013861398</td>
<td>18</td>
<td>18</td>
<td>0.75053</td>
<td>0.75053</td>
<td>0.75053</td>
</tr>
<tr>
<td>6</td>
<td>1326029789</td>
<td>417047966</td>
<td>18</td>
<td>0.48211</td>
<td>0.57102</td>
<td>0.48211</td>
</tr>
<tr>
<td>7</td>
<td>932142747</td>
<td>850344656</td>
<td>18</td>
<td>0.83457</td>
<td>0.92566</td>
<td>0.83457</td>
</tr>
<tr>
<td>8</td>
<td>1988843719</td>
<td>2067665501</td>
<td>18</td>
<td>0.07674</td>
<td>0.16730</td>
<td>0.07674</td>
</tr>
<tr>
<td>9</td>
<td>516966271</td>
<td>607886093</td>
<td>18</td>
<td>1.42407</td>
<td>0.51513</td>
<td>1.42407</td>
</tr>
<tr>
<td>10</td>
<td>2137808851</td>
<td>1550198721</td>
<td>18</td>
<td>0.00452</td>
<td>0.91543</td>
<td>0.00452</td>
</tr>
</tbody>
</table>

Changing Seed_2 for the CALL RANEXP statement, when I=5, forces the stream of the variates for X2 to deviate from the stream of the variates for X1. Changing Seed_3 on the RANEXP function, however, has no effect.

**See Also**

Function:

“RANEXP Function” on page 777

---

**CALL RANGAM Routine**

Returns a random variate from a gamma distribution

**Category:** Random Number

**Syntax**

CALL RANGAM(seed,a,x);

**Arguments**

*seed*

is the seed value. A new value for *seed* is returned each time CALL RANGAM is executed.

**Range:** seed < 2^{31} - 1

**Note:** If seed ≤ 0, the time of day is used to initialize the seed stream.

**See:** “Seed Values” on page 257 for more information about seed values

*a*

is a numeric shape parameter.
Range: $a > 0$

$x$

is a numeric variable. A new value for the random variate $x$ is returned each time CALL RANGAM is executed.

Details

The CALL RANGAM routine updates seed and returns a variate $x$ that is generated from a gamma distribution with parameter $a$.

By adjusting the seeds, you can force streams of variates to agree or disagree for some or all of the observations in the same, or in subsequent, DATA steps.

For $a > 1$, an acceptance-rejection method by Cheng is used (Cheng, 1977; see in “References” on page 928). For $a \leq 1$, an acceptance-rejection method by Fishman is used (Fishman, 1978; see in “References” on page 928).

Comparisons

The CALL RANGAM routine gives greater control of the seed and random number streams than does the RANGAM function.

Examples

This example uses the CALL RANGAM routine:

```plaintext
options nodate pageno=1 linesize=80 pagesize=60;

data case;
  retain Seed_1 Seed_2 Seed_3 45;
  a=2;
  do i=1 to 10;
    call rangam(Seed_1,a,X1);
    call rangam(Seed_2,a,X2);
    X3=rangam(Seed_3,a);
    if i=5 then
      do;
        Seed_2=18;
        Seed_3=18;
      end;
      output;
    end;
run;

proc print;
  id i;
  var Seed_1-Seed_3 X1-X3;
run;
```

The following output shows the results:
Output 4.9  The RANGAM Example

<table>
<thead>
<tr>
<th>i</th>
<th>Seed_1</th>
<th>Seed_2</th>
<th>Seed_3</th>
<th>X1</th>
<th>X2</th>
<th>X3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1404437564</td>
<td>1404437564</td>
<td>45</td>
<td>1.30569</td>
<td>1.30569</td>
<td>1.30569</td>
</tr>
<tr>
<td>2</td>
<td>1326029789</td>
<td>1326029789</td>
<td>45</td>
<td>1.87514</td>
<td>1.87514</td>
<td>1.87514</td>
</tr>
<tr>
<td>3</td>
<td>1988843719</td>
<td>1988843719</td>
<td>45</td>
<td>1.71597</td>
<td>1.71597</td>
<td>1.71597</td>
</tr>
<tr>
<td>4</td>
<td>50049159</td>
<td>50049159</td>
<td>45</td>
<td>1.59304</td>
<td>1.59304</td>
<td>1.59304</td>
</tr>
<tr>
<td>5</td>
<td>802575599</td>
<td>18</td>
<td>18</td>
<td>0.43342</td>
<td>0.43342</td>
<td>0.43342</td>
</tr>
<tr>
<td>6</td>
<td>100573943</td>
<td>991271755</td>
<td>18</td>
<td>1.11812</td>
<td>1.32646</td>
<td>1.11812</td>
</tr>
<tr>
<td>7</td>
<td>1986749826</td>
<td>1437043694</td>
<td>18</td>
<td>0.68415</td>
<td>0.88806</td>
<td>0.68415</td>
</tr>
<tr>
<td>8</td>
<td>52428589</td>
<td>959908645</td>
<td>18</td>
<td>1.62296</td>
<td>2.46091</td>
<td>1.62296</td>
</tr>
<tr>
<td>9</td>
<td>1216356463</td>
<td>1225034217</td>
<td>18</td>
<td>2.26455</td>
<td>4.06596</td>
<td>2.26455</td>
</tr>
<tr>
<td>10</td>
<td>805366679</td>
<td>425626811</td>
<td>18</td>
<td>2.16723</td>
<td>6.94703</td>
<td>2.16723</td>
</tr>
</tbody>
</table>

Changing Seed_2 for the CALL RANGAM statement, when I=5, forces the stream of the variates for X2 to deviate from the stream of the variates for X1. Changing Seed_3 on the RANGAM function, however, has no effect.

See Also

Function:
“RANGAM Function” on page 778

CALL RANNOR Routine

Returns a random variate from a normal distribution

Category:  Random Number

Syntax

CALL RANNOR(seed,x);

Arguments

seed
is the seed value. A new value for seed is returned each time CALL RANNOR is executed.

Range:  seed < 2^31 - 1

Note:  If seed ≤ 0, the time of day is used to initialize the seed stream.

See:  “Seed Values” on page 257 for more information about seed values

x
is a numeric variable. A new value for the random variate x is returned each time CALL RANNOR is executed.
Details

The CALL RANNOR routine updates seed and returns a variate \(x\) that is generated from a normal distribution, with mean 0 and variance 1.

By adjusting the seeds, you can force streams of variates to agree or disagree for some or all of the observations in the same, or in subsequent, DATA steps.

The CALL RANNOR routine uses the Box-Muller transformation of RANUNI uniform variates.

Comparisons

The CALL RANNOR routine gives greater control of the seed and random number streams than does the RANNOR function.

Examples

This example uses the CALL RANNOR routine:

```sas
options nodate pageno=1 linesize=80 pagesize=60;

data case;
  retain Seed_1 Seed_2 Seed_3 45;
  do i=1 to 10;
    call rannor(Seed_1,X1);
    call rannor(Seed_2,X2);
    X3=rannor(Seed_3);
    if i=5 then
      do;
        Seed_2=18;
        Seed_3=18;
      end;
    output;
  end;
run;

proc print;
  id i;
  var Seed_1-Seed_3 X1-X3;
run;
```

The following output shows the results:

<table>
<thead>
<tr>
<th>i</th>
<th>Seed_1</th>
<th>Seed_2</th>
<th>Seed_3</th>
<th>X1</th>
<th>X2</th>
<th>X3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1404437564</td>
<td>1404437564</td>
<td>45</td>
<td>-0.85252</td>
<td>-0.85252</td>
<td>-0.85252</td>
</tr>
<tr>
<td>2</td>
<td>1445125588</td>
<td>1445125588</td>
<td>45</td>
<td>-0.05865</td>
<td>-0.05865</td>
<td>-0.05865</td>
</tr>
<tr>
<td>3</td>
<td>1326029789</td>
<td>1326029789</td>
<td>45</td>
<td>-0.90628</td>
<td>-0.90628</td>
<td>-0.90628</td>
</tr>
<tr>
<td>4</td>
<td>1988843719</td>
<td>1988843719</td>
<td>45</td>
<td>1.15526</td>
<td>1.15526</td>
<td>1.15526</td>
</tr>
<tr>
<td>5</td>
<td>2137808851</td>
<td>18</td>
<td>18</td>
<td>1.68697</td>
<td>1.68697</td>
<td>1.68697</td>
</tr>
<tr>
<td>6</td>
<td>1233028129</td>
<td>991271755</td>
<td>18</td>
<td>-0.47276</td>
<td>-1.44726</td>
<td>-0.47276</td>
</tr>
<tr>
<td>7</td>
<td>50049159</td>
<td>1437043694</td>
<td>18</td>
<td>1.33423</td>
<td>-0.87677</td>
<td>1.33423</td>
</tr>
<tr>
<td>8</td>
<td>802575599</td>
<td>959908645</td>
<td>18</td>
<td>-1.63511</td>
<td>-0.97261</td>
<td>-1.63511</td>
</tr>
<tr>
<td>9</td>
<td>100573943</td>
<td>1225034217</td>
<td>18</td>
<td>1.55410</td>
<td>-0.64742</td>
<td>1.55410</td>
</tr>
<tr>
<td>10</td>
<td>414117170</td>
<td>425626811</td>
<td>18</td>
<td>0.10736</td>
<td>0.14963</td>
<td>0.10736</td>
</tr>
</tbody>
</table>

Output 4.10  The RANNOR Example
Changing Seed_2 for the CALL RANNOR statement, when \( I=5 \), forces the stream of the variates for \( X_2 \) to deviate from the stream of the variates for \( X_1 \). Changing Seed_3 on the RANNOR function, however, has no effect.

**See Also**

Function:

“RANNOR Function” on page 780

---

**CALL RANPERK Routine**

Randomly permutes the values of the arguments, and returns a permutation of \( k \) out of \( n \) values

**Category:** Random Number

**Syntax**

```
CALL RANPERK(seed, k, variable-1<, variable-2, ...>);
```

**Arguments**

- **seed**
  - is a numeric variable that contains the random number seed. For more information about seeds, see “Seed Values” on page 257.

- **\( k \)**
  - is the number of values that you want to have in the random permutation.

- **variable**
  - specifies all numeric variables, or all character variables that have the same length. \( K \) values of these variables are randomly permuted.

**Examples**

The following example shows how to generate random permutations of given values by using the CALL RANPERK routine.

```sas
data _null_;
  array x x1-x5 {1 2 3 4 5};
  seed = 1234567890123;
  do n=1 to 10;
    call ranperk(seed, 3, of x1-x5);
    put seed= @20 ' x=' x1-x3;
  end;
run;
```

The following lines are written to the SAS log:

- seed=1332351321   x= 5 4 2
- seed=829042065   x= 4 1 3
CALL RANPERM Routine

Randomly permutes the values of the arguments

Category: Random Number

Syntax

CALL RANPERM(seed, variable-1<, variable-2, ...>);

Arguments

seed
is a numeric variable that contains the random number seed. For more information about seeds, see “Seed Values” on page 257.

variable
specifies all numeric variables or all character variables that have the same length. The values of these variables are randomly permuted.

Examples

The following example generates random permutations of given values by using the CALL RANPERM routine.

```plaintext
data _null_; 
  array x x1-x4 (1 2 3 4); 
  seed = 1234567890123; 
  do n=1 to 10; 
    call ranperm(seed, of x1-x4); 
    put seed= $20 ' x=' x1-x4; 
  end; 
run;
```

See Also

Functions:

“CALL ALLPERM Routine” on page 335
“CALL RANPERM Routine” on page 376
The following lines are written to the SAS log:

seed=1332351321    x= 1 3 2 4
seed=829042065     x= 3 4 2 1
seed=767738639     x= 4 2 3 1
seed=1280236105    x= 1 2 4 3
seed=670350431     x= 2 1 4 3
seed=1956939964    x= 2 4 3 1
seed=353939815     x= 4 1 2 3
seed=1996660805    x= 4 3 1 2
seed=1835940555    x= 4 3 2 1
seed=910897519     x= 3 2 1 4

See Also

CALL Routines:
“CALL ALLPERM Routine” on page 335
“CALL RANPERK Routine” on page 375

CALL RANPOI Routine

Returns a random variate from a Poisson distribution

Category: Random Number

Syntax

CALL RANPOI(seed,m,x);

Arguments

*seed*

is the seed value. A new value for seed is returned each time CALL RANPOI is executed.

Range: seed < 2^{31} - 1

Note: If seed ≤ 0, the time of day is used to initialize the seed stream.

See: “Seed Values” on page 257 for more information about seed values

*m*

is a numeric mean parameter.

Range: m ≥ 0

*x*

is a numeric variable. A new value for the random variate x is returned each time CALL RANPOI is executed.
Details

The CALL RANPOI routine updates seed and returns a variate x that is generated from a Poisson distribution, with mean m.

By adjusting the seeds, you can force streams of variates to agree or disagree for some or all of the observations in the same, or in subsequent, DATA steps.

For m < 85, an inverse transform method applied to a RANUNI uniform variate is used (Fishman, 1976; see in “References” on page 928). For m ≥ 85, the normal approximation of a Poisson random variable is used. To expedite execution, internal variables are calculated only on initial calls (that is, with each new m).

Comparisons

The CALL RANPOI routine gives greater control of the seed and random number streams than does the RANPOI function.

Examples

This example uses the CALL RANPOI routine:

```plaintext
options nodate pageno=1 linesize=80 pagesize=60;

data case;
  retain Seed_1 Seed_2 Seed_3 45;
  m=120;
  do i=1 to 10;
    call ranpoi(Seed_1,m,X1);
    call ranpoi(Seed_2,m,X2);
    X3=ranpoi(Seed_3,m);
    if i=5 then
      do;
        Seed_2=18;
        Seed_3=18;
      end;
    output;
  end;
run;

proc print;
  id i;
  var Seed_1-Seed_3 X1-X3;
run;
```
The following output shows the results:

Output 4.11  The RANPOI Example

<table>
<thead>
<tr>
<th>i</th>
<th>Seed_1</th>
<th>Seed_2</th>
<th>Seed_3</th>
<th>X1</th>
<th>X2</th>
<th>X3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1404437564</td>
<td>1404437564</td>
<td></td>
<td>45</td>
<td>111</td>
<td>111</td>
</tr>
<tr>
<td>2</td>
<td>1445125588</td>
<td>1445125588</td>
<td></td>
<td>45</td>
<td>119</td>
<td>119</td>
</tr>
<tr>
<td>3</td>
<td>1326029789</td>
<td>1326029789</td>
<td></td>
<td>45</td>
<td>110</td>
<td>110</td>
</tr>
<tr>
<td>4</td>
<td>1988843719</td>
<td>1988843719</td>
<td></td>
<td>45</td>
<td>133</td>
<td>133</td>
</tr>
<tr>
<td>5</td>
<td>2137808851</td>
<td>18</td>
<td>18</td>
<td>138</td>
<td>138</td>
<td>138</td>
</tr>
<tr>
<td>6</td>
<td>1233028129</td>
<td>991271755</td>
<td>18</td>
<td>115</td>
<td>104</td>
<td>115</td>
</tr>
<tr>
<td>7</td>
<td>50049159</td>
<td>1437043694</td>
<td>18</td>
<td>135</td>
<td>110</td>
<td>135</td>
</tr>
<tr>
<td>8</td>
<td>802575599</td>
<td>959908645</td>
<td>18</td>
<td>102</td>
<td>109</td>
<td>102</td>
</tr>
<tr>
<td>9</td>
<td>100573943</td>
<td>1225034217</td>
<td>18</td>
<td>137</td>
<td>113</td>
<td>137</td>
</tr>
<tr>
<td>10</td>
<td>414117170</td>
<td>426626811</td>
<td>18</td>
<td>121</td>
<td>122</td>
<td>121</td>
</tr>
</tbody>
</table>

Changing Seed_2 for the CALL RANPOI statement, when I=5, forces the stream of the variates for X2 to deviate from the stream of the variates for X1. Changing Seed_3 on the RANPOI function, however, has no effect.

See Also

Function:
“RANPOI Function” on page 781

CALL RANTBL Routine

Returns a random variate from a tabled probability distribution

Category:  Random Number

Syntax

CALL RANTBL(seed,p1,...,pi,...,pn,x);

Arguments

seed
is the seed value. A new value for seed is returned each time CALL RANTBL is executed.

Range:  seed < 2^31 - 1

Note:  If seed ≤ 0, the time of day is used to initialize the seed stream.

See:  “Seed Values” on page 257 for more information about seed values

p,
is a numeric SAS value.
Range: $0 \leq p \leq 1$ for $0 < i \leq n$

$x$

is a numeric SAS variable. A new value for the random variate $x$ is returned each time CALL RANTBL is executed.

Details

The CALL RANTBL routine updates seed and returns a variate $x$ generated from the probability mass function defined by $p_i$ through $p_n$.

By adjusting the seeds, you can force streams of variates to agree or disagree for some or all of the observations in the same, or in subsequent, DATA steps.

An inverse transform method applied to a RANUNI uniform variate is used. The CALL RANTBL routine returns these data:

1   with probability $p_1$
2   with probability $p_2$
.   
.   
n   with probability $p_n$
n + 1 with probability $1 - \sum_{i=1}^{n} p_i$ if $\sum_{i=1}^{n} p_i \leq 1$

If, for some index $j < n$,

$$\sum_{i=1}^{j} p_i \geq 1$$

RANTBL returns only the indices 1 through $j$, with the probability of occurrence of the index $j$ equal to

$$1 - \sum_{i=1}^{j-1} p_i$$

Comparisons

The CALL RANTBL routine gives greater control of the seed and random number streams than does the RANTBL function.

Examples

This example uses the CALL RANTBL routine:

```sas
options nodate pageno=1 linesize=80 pagesize=60;

data case;
```

retain Seed_1 Seed_2 Seed_3 45;
input p1-p9;
do i=1 to 10;
call rantbl(Seed_1,of p1-p9,X1);
call rantbl(Seed_2,of p1-p9,X2);
X3=rantbl(Seed_3,of p1-p9);
if i=5 then 
do;
  Seed_2=18;
  Seed_3=18;
end;
output;
end;
datalines;
.02 .04 .06 .08 .1 .12 .14 .16 .18;
proc print;
id i;
  var Seed_1-Seed_3 X1-X3;
run;
The following output shows the results:

<table>
<thead>
<tr>
<th>i</th>
<th>Seed_1</th>
<th>Seed_2</th>
<th>Seed_3</th>
<th>X1</th>
<th>X2</th>
<th>X3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>694315054</td>
<td>694315054</td>
<td>45</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>1404437564</td>
<td>1404437564</td>
<td>45</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>2130505156</td>
<td>2130505156</td>
<td>45</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>1445125588</td>
<td>1445125588</td>
<td>45</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>1013861398</td>
<td>18</td>
<td>18</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>1326029789</td>
<td>707222751</td>
<td>18</td>
<td>8</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>932142347</td>
<td>991271355</td>
<td>18</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>1988843719</td>
<td>422705333</td>
<td>18</td>
<td>10</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>516966271</td>
<td>1437043694</td>
<td>18</td>
<td>5</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>2137808851</td>
<td>1264538018</td>
<td>18</td>
<td>10</td>
<td>8</td>
<td>10</td>
</tr>
</tbody>
</table>

Changing Seed_2 for the CALL RANTBL statement, when I=5, forces the stream of variates for X2 to deviate from the stream of variates for X1. Changing Seed_3 on the RANTBL function, however, has no effect.

See Also

Function:
“RANTBL Function” on page 782

CALL RANTRI Routine

Returns a random variate from a triangular distribution
CALL RANTRI Routine

Chapter 4

Category: Random Number

Syntax

CALL RANTRI(seed,h,x);

Arguments

seed
is the seed value. A new value for seed is returned each time CALL RANTRI is executed.

Range: seed < 2^{31} - 1

Note: If seed ≤ 0, the time of day is used to initialize the seed stream.

See: “Seed Values” on page 257 for more information about seed values

h
is a numeric SAS value.

Range: 0 < h < 1

x
is a numeric SAS variable. A new value for the random variate x is returned each time CALL RANTRI is executed.

Details

The CALL RANTRI routine updates seed and returns a variate x generated from a triangular distribution on the interval (0,1) with parameter h, which is the modal value of the distribution.

By adjusting the seeds, you can force streams of variates to agree or disagree for some or all of the observations in the same, or in subsequent, DATA steps.

The CALL RANTRI routine uses an inverse transform method applied to a RANUNI uniform variate.

Comparisons

The CALL RANTRI routine gives greater control of the seed and random number streams than does the RANTRI function.

Examples

This example uses the CALL RANTRI routine:

```sas
options nodate pageno=1 linesize=80 pagesize=60;

data case;
  retain Seed_1 Seed_2 Seed_3 45;
  h=.2;
  do i=1 to 10;
    call rantri(Seed_1,h,X1);
    call rantri(Seed_2,h,X2);
    X3=rantri(Seed_3,h);
    if i=5 then
```
do;
  Seed_2=18;
  Seed_3=18;
end;
output;
end;
run;

proc print;
  id i;
  var Seed_1-Seed_3 X1-X3;
run;

The following output shows the results:

Output 4.13  The RANTRI Example

<table>
<thead>
<tr>
<th>i</th>
<th>Seed_1</th>
<th>Seed_2</th>
<th>Seed_3</th>
<th>X1</th>
<th>X2</th>
<th>X3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>694315054</td>
<td>694315054</td>
<td>45</td>
<td>0.26424</td>
<td>0.26424</td>
<td>0.26424</td>
</tr>
<tr>
<td>2</td>
<td>1404437564</td>
<td>1404437564</td>
<td>45</td>
<td>0.47388</td>
<td>0.47388</td>
<td>0.47388</td>
</tr>
<tr>
<td>3</td>
<td>2130505156</td>
<td>2130505156</td>
<td>45</td>
<td>0.92047</td>
<td>0.92047</td>
<td>0.92047</td>
</tr>
<tr>
<td>4</td>
<td>1445125588</td>
<td>1445125588</td>
<td>45</td>
<td>0.48848</td>
<td>0.48848</td>
<td>0.48848</td>
</tr>
<tr>
<td>5</td>
<td>1013861398</td>
<td>18</td>
<td>18</td>
<td>0.35015</td>
<td>0.35015</td>
<td>0.35015</td>
</tr>
<tr>
<td>6</td>
<td>1326029789</td>
<td>707222751</td>
<td>18</td>
<td>0.44681</td>
<td>0.26751</td>
<td>0.44681</td>
</tr>
<tr>
<td>7</td>
<td>932142747</td>
<td>991271755</td>
<td>18</td>
<td>0.32713</td>
<td>0.34371</td>
<td>0.32713</td>
</tr>
<tr>
<td>8</td>
<td>1988843719</td>
<td>422705333</td>
<td>18</td>
<td>0.75690</td>
<td>0.19841</td>
<td>0.75690</td>
</tr>
<tr>
<td>9</td>
<td>516966271</td>
<td>1437043694</td>
<td>18</td>
<td>0.22063</td>
<td>0.48555</td>
<td>0.22063</td>
</tr>
<tr>
<td>10</td>
<td>2137808851</td>
<td>1264538018</td>
<td>18</td>
<td>0.93997</td>
<td>0.42648</td>
<td>0.93997</td>
</tr>
</tbody>
</table>

Changing Seed_2 for the CALL RANTRI statement, when I=5, forces the stream of the variates for X2 to deviate from the stream of the variates for X1. Changing Seed_3 on the RANTRI function has, however, no effect.

See Also

Function:
  “RANTRI Function” on page 783

CALL RANUNI Routine

Returns a random variate from a uniform distribution

Category:  Random Number

Syntax

CALL RANUNI(seed,x);
Arguments

*seed*

is the seed value. A new value for *seed* is returned each time CALL RANUNI is executed.

**Range:**  $seed < 2^{31} - 1$

**Note**  If $seed \leq 0$, the time of day is used to initialize the seed stream.

**See:** “Seed Values” on page 257 for more information about seed values

*x*

is a numeric variable. A new value for the random variate *x* is returned each time CALL RANUNI is executed.

Details

The CALL RANUNI routine updates *seed* and returns a variate *x* that is generated from the uniform distribution on the interval (0,1), using a prime modulus multiplicative generator with modulus $2^{31} - 1$ and multiplier 397204094 (Fishman and Moore 1982) (See “References” on page 928).

By adjusting the seeds, you can force streams of variates to agree or disagree for some or all of the observations in the same, or in subsequent, DATA steps.

Comparisons

The CALL RANUNI routine gives greater control of the seed and random number streams than does the RANUNI function.

Examples

This example uses the CALL RANUNI routine:

```plaintext
options nodate pageno=1 linesize=80 pagesize=60;

data case;
  retain Seed_1 Seed_2 Seed_3 45;
  do i=1 to 10;
    call ranuni(Seed_1,X1);
    call ranuni(Seed_2,X2);
    X3=ranuni(Seed_3);
    if i=5 then
      do;
        Seed_2=18;
        Seed_3=18;
      end;
    output;
  end;
run;

proc print;
  id i;
  var Seed_1-Seed_3 X1-X3;
run;
```
The following output shows the results:

Output 4.14  The RANUNI Example

<table>
<thead>
<tr>
<th>i</th>
<th>Seed_1</th>
<th>Seed_2</th>
<th>Seed_3</th>
<th>X1</th>
<th>X2</th>
<th>X3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>694315054</td>
<td>694315054</td>
<td>45</td>
<td>0.32332</td>
<td>0.32332</td>
<td>0.32332</td>
</tr>
<tr>
<td>2</td>
<td>1404437564</td>
<td>1404437564</td>
<td>45</td>
<td>0.65399</td>
<td>0.65399</td>
<td>0.65399</td>
</tr>
<tr>
<td>3</td>
<td>2130505156</td>
<td>2130505156</td>
<td>45</td>
<td>0.99209</td>
<td>0.99209</td>
<td>0.99209</td>
</tr>
<tr>
<td>4</td>
<td>1445125588</td>
<td>1445125588</td>
<td>45</td>
<td>0.67294</td>
<td>0.67294</td>
<td>0.67294</td>
</tr>
<tr>
<td>5</td>
<td>1013861398</td>
<td>18</td>
<td>18</td>
<td>0.47212</td>
<td>0.47212</td>
<td>0.47212</td>
</tr>
<tr>
<td>6</td>
<td>1326029789</td>
<td>707222751</td>
<td>18</td>
<td>0.61748</td>
<td>0.32933</td>
<td>0.61748</td>
</tr>
<tr>
<td>7</td>
<td>932142747</td>
<td>991271755</td>
<td>18</td>
<td>0.43406</td>
<td>0.46160</td>
<td>0.43406</td>
</tr>
<tr>
<td>8</td>
<td>1988843719</td>
<td>422705333</td>
<td>18</td>
<td>0.92613</td>
<td>0.19684</td>
<td>0.92613</td>
</tr>
<tr>
<td>9</td>
<td>516966271</td>
<td>1437043694</td>
<td>18</td>
<td>0.24073</td>
<td>0.66918</td>
<td>0.24073</td>
</tr>
<tr>
<td>10</td>
<td>2137808851</td>
<td>1264538018</td>
<td>18</td>
<td>0.99549</td>
<td>0.58885</td>
<td>0.99549</td>
</tr>
</tbody>
</table>

Changing Seed_2 for the CALL RANUNI statement, when I=5, forces the stream of the variates for X2 to deviate from the stream of the variates for X1. Changing Seed_3 on the RANUNI function, however, has no effect.

See Also

Function:
“RANUNI Function” on page 784

---

**CALL RXCHANGE Routine**

Changes one or more substrings that match a pattern

**Category:** Character String Matching  
**Restriction:** Use with the RXPARSE function

**Syntax**

CALL RXCHANGE (rx, times, old-string,<, new-string>);

**Arguments**

- **rx**
  specifies a numeric value that is returned from the RXPARSE function.  
  **Tip:** The value of `rx` points to an expression that is parsed by RXPARSE. CALL RXCHANGE uses the expression to find and change a matching substring.

- **times**
  is a numeric value that specifies the maximum number of times to change matching substrings.
Restriction:  Maximum value of times is 2,147,483,647.

old-string
specifies the character expression to be searched and changed as a result of the change operation.

new-string
is a character variable that contains the result of the change operation.

Details
If old-string is a character variable, new-string is optional. If new-string is omitted, old-string is changed in place.
If old-string is an expression that is not a character variable, you must specify new-string.

Comparisons
The regular expression (RX) functions and CALL routines work together to manipulate strings that match patterns. Use the RXPARSE function to parse a pattern you specify. Use the RXMATCH function and the CALL RXCHANGE and CALL RXSUBSTR routines to match or modify your data. Use the CALL RXFREE routine to free allocated space.

Example

See the RXPARSE function “Examples” on page 812.

See Also

Functions and CALL routines:
“CALL RXFREE Routine” on page 386
“CALL RXSUBSTR Routine” on page 387
“RXMATCH Function” on page 801
“RXPARSE Function” on page 802

CALL RXFREE Routine

Frees memory allocated by other regular expression (RX) functions and CALL routines

Category:  Character String Matching
Restriction:  Use with the RXPARSE function

Syntax
CALL RXFREE (rx);
Arguments

rx
specifies a numeric value that is returned from the RXPARSE function.

Comparisons
The regular expression (RX) functions and CALL routines work together to manipulate strings that match patterns. Use the RXPARSE function to parse a pattern you specify. Use the RXMATCH function and the CALL RXCHANGE and CALL RXSUBSTR routines to match or modify your data. Use the CALL RXFREE routine to free allocated space.

Example
See the RXPARSE function “Examples” on page 812.

See Also
Functions and CALL routines:
“CALL RXCHANGE Routine” on page 385
“CALL RXSUBSTR Routine” on page 387
“RXMATCH Function” on page 801
“RXPARSE Function” on page 802

CALL RXSUBSTR Routine

Finds the position, length, and score of a substring that matches a pattern

Category:  Character String Matching
Restriction:  Use with the RXPARSE function

Syntax
CALL RXSUBSTR (rx, string, position);
CALL RXSUBSTR (rx, string, position, length);
CALL RXSUBSTR (rx, string, position, length, score);

Arguments

rx
specifies a numeric value that is returned from the RXPARSE function.
**string**
specifies the character expression to be searched.

**position**
specifies a numeric position in a string where the substring that is matched by the pattern begins. If there is no match, the result is zero.

**length**
specifies a numeric value that is the length of the substring that is matched by the pattern.

**score**
specifies an integer value based on the number of matches for a particular pattern in a substring.

**Details**
CALL RXSUBSTR searches the variable string for the pattern from RXPARSE, returns the position of the start of the string, indicates the length of the matched string, and returns a score value. By default, when a pattern matches more than one substring beginning at a specific position, the longest substring is selected.

**Comparisons**
CALL RXSUBSTR performs the same matching as RXMATCH, but CALL RXSUBSTR additionally allows you to use the length and score arguments to receive more information about the match. The regular expression (RX) functions and CALL routines work together to manipulate strings that match patterns. Use the RXPARSE function to parse a pattern you specify. Use the RXMATCH function and the CALL RXCHANGE and CALL RXSUBSTR routines to match or modify your data. Use the CALL RXFREE routine to free allocated space.

**Example**

See the RXPARSE function “Examples” on page 812.

**See Also**

Functions and CALL routines:
“CALL RXCHANGE Routine” on page 385
“CALL RXFREE Routine” on page 386
“RXMATCH Function” on page 801
“RXPARSE Function” on page 802

---

**CALL SCAN Routine**

Returns the position and length of a given word from a character expression

**Category:** Character
Syntax
CALL SCAN(string, n, position, length <,delimiters>);

Arguments

string
specifies a character constant, variable, or expression.

n
is a numeric constant, variable, or expression that specifies the number of the word in the character string that you want the CALL SCAN routine to select. The following rules apply:
- If n is positive, CALL SCAN counts words from left to right.
- If n is negative, CALL SCAN counts words from right to left.
- If n is zero, or |n| is greater than the number of words in the character string, CALL SCAN returns a position and length of zero.

position
specifies a numeric variable in which the position of the word is returned.

length
specifies a numeric variable in which the length of the word is returned.

delimiters
is a character constant, variable, or expression that specifies the characters that you want CALL SCAN to use to separate words in the character string.

Default: If you omit delimiters in an ASCII environment, SAS uses the following characters:
- blank . < ( + & ! $ * ); ^ – / , % |

In ASCII environments without the ^ character, SCAN uses the ~ character instead.
If you omit delimiters in an EBCDIC environment, SAS uses the following characters:
- blank . < ( + | & ! $ * ); ~ – / , % |

tip: If you represent delimiters as a constant, enclose delimiters in quotation marks.

Details
In the context of the CALL SCAN routine, “word” refers to a substring that
- is bounded on the left by a delimiter or the beginning of the string
- is bounded on the right by a delimiter or the end of the string
- contains no delimiters.

Delimiters that are located before the first word or after the last word in the character string do not affect the CALL SCAN routine. If two or more contiguous delimiters exist, CALL SCAN treats them as one.
To retrieve the designated word as a character string after the call to CALL SCAN, use the following function:
substrn(string, position, length)
Comparisons
The CALL SCANQ routine ignores delimiters that are enclosed in quotation marks. CALL SCAN does not.

Examples
The following example shows how you can use the CALL SCAN routine.

```sas
options pageno=1 nodate ls=80 ps=64;

data artists;
  input string $60.;
  drop string;
  do i=1 to 99;
    call scan(string, i, position, length);
    if not position then leave;
    Name=substrn(string, position, length);
    output;
  end;
  datalines;
Picasso Toulouse-Lautrec Turner "Van Gogh" Velazquez
;

proc print data=artists;
run;
```

Output 4.15  SAS Output from the CALL SCAN Routine

<table>
<thead>
<tr>
<th>Obs</th>
<th>i</th>
<th>position</th>
<th>length</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>Picasso</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>9</td>
<td>8</td>
<td>Toulouse</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>18</td>
<td>7</td>
<td>Lautrec</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>26</td>
<td>6</td>
<td>Turner</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>33</td>
<td>4</td>
<td>&quot;Van</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>38</td>
<td>5</td>
<td>Gogh</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>44</td>
<td>9</td>
<td>Velazquez</td>
</tr>
</tbody>
</table>

See Also

Functions and CALL Routines:
“SCAN Function” on page 816
“SCANQ Function” on page 817
“CALL SCANQ Routine” on page 390

CALL SCANQ Routine

Returns the position and length of a given word from a character expression, and ignores delimiters that are enclosed in quotation marks
Category: Character

Syntax

CALL SCANQ(string, n, position, length <.delimiters>);

Arguments

string
 specifies a character constant, variable, or expression.

n
 is a numeric constant, variable, or expression that specifies the number of the word in the character string that you want the CALL SCANQ routine to select. The following rules apply:

- If \( n \) is positive, CALL SCANQ counts words from left to right.
- If \( n \) is negative, CALL SCANQ counts words from right to left.
- If \( n \) is zero, or \(|n|\) is greater than the number of words in the character string, CALL SCANQ returns a position and length of zero.

position
 specifies a numeric variable in which the position of the word is returned.

length
 specifies a numeric variable in which the length of the word is returned.

delimiters
 is a character constant, variable, or expression that specifies the characters that you want CALL SCANQ to use to separate words in the character string.

Default: If you omit delimiters CALL SCANQ uses white space characters (blank, horizontal and vertical tab, carriage return, line feed, and form feed) as delimiters.

Restriction: You cannot use single or double quotation marks as delimiters.

Details

In the context of the CALL SCANQ routine, “word” refers to a substring that

- is bounded on the left by a delimiter or the beginning of the string
- is bounded on the right by a delimiter or the end of the string
- contains no delimiters except those that are enclosed in quotation marks.

If the value of the character string contains quotation marks, CALL SCANQ ignores delimiters inside the strings in quotation marks. If the value of the character string contains unmatched quotation marks, then scanning from left to right will produce different words than scanning from right to left.

Delimiters that are located before the first word or after the last word in the character string do not affect the CALL SCANQ routine. If two or more contiguous delimiters exist, CALL SCANQ treats them as one.

To retrieve the designated word as a character string after the call to CALL SCANQ, use the following function:

\[ \text{substrn(string, position, length)} \]
Comparisons
The CALL SCANQ routine ignores delimiters that are enclosed in quotation marks. CALL SCAN does not.

Examples
The following example shows how you can use the CALL SCANQ routine.

```sas
options pageno=1 nodate ls=80 ps=64;

data artists2;
  input string $60.;
  drop string;
  do i=1 to 99;
    call scanq(string, i, position, length);
    if not position then leave;
    Name=substrn(string, position, length);
    output;
  end;
  datalines;
Picasso Toulouse-Lautrec Turner "Van Gogh" Velazquez
;
proc print data=artists2;
run;
```

Output 4.16  SAS Output from the CALL SCANQ Routine

<table>
<thead>
<tr>
<th>The SAS System</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obs</td>
<td>i</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

See Also

Functions and CALL Routines:
“SCAN Function” on page 816
“SCANQ Function” on page 817
“CALL SCAN Routine” on page 388

CALL SET Routine

Links SAS data set variables to DATA step or macro variables that have the same name and data type
Category: Variable Control

**Syntax**

CALL SET(data-set-id);

**Arguments**

*data-set-id*

is the identifier that is assigned by the OPEN function when the data set is opened.

**Details**

Using SET can significantly reduce the coding that is required for accessing variable values for modification or verification when you use functions to read or to manipulate a SAS file. After a CALL SET, whenever a read is performed from the SAS data set, the values of the corresponding macro or DATA step variables are set to the values of the matching SAS data set variables. If the variable lengths do not match, the values are truncated or padded according to need. If you do not use SET, then you must use the GETVARC and GETVARN functions to move values explicitly between data set variables and macro or DATA step variables.

As a general rule, use CALL SET immediately following OPEN if you want to link the data set and the macro and DATA step variables.

**Examples**

This example uses the CALL SET routine:

- The following statements automatically set the values of the macro variables PRICE and STYLE when an observation is fetched:

```sas
%macro setvar;
  %let dsid=%sysfunc(open(sasuser.houses,i));
  /* No leading ampersand with %SYSCALL */
  %syscall set(dsid);
  %let rc=%sysfunc(fetchobs(&dsid,10));
  %let rc=%sysfunc(close(&dsid));
%mend setvar;

%global price style;
%setvar
%put _global_;
```

- The %PUT statement writes these lines to the SAS log:

```
GLOBAL PRICE 127150
GLOBAL STYLE CONDO
```

- The following statements obtain the values for the first 10 observations in SASUSER.HOUSES and store them in MYDATA:

```sas
data mydata;
  /* create variables for assignment */
  /*by CALL SET */
  length style $8 sqfeet bedrooms baths 8
```
CALL SLEEP Routine

Suspends the execution of a program that invokes this call routine for a specified period of time

Category: Special
See: CALL SLEEP Routine in the documentation for your operating environment.

Syntax

CALL SLEEP(n<, unit>)

Arguments

n
is a numeric constant that specifies the number of units of time for which you want to suspend execution of a program.
Range: \( n \geq 0 \)

unit
specifies the unit of time, as a power of 10, which is applied to \( n \). For example, 1 corresponds to a second, and .001 corresponds to a millisecond.
Default: .001

Details

The CALL SLEEP routine suspends the execution of a program that invokes this call routine for a period of time that you specify. The program can be a DATA step, macro, IML, SCL, or anything that can invoke a call routine. The maximum sleep period for the CALL SLEEP routine is 46 days.
Examples

Example 1: Suspending Execution for a Specified Period of Time

The following example tells SAS to suspend the execution of the DATA step PAYROLL for 1 minute and 10 seconds:

```sas
data payroll;
    call sleep(7000,.01);
    ...more SAS statements...
run;
```

Example 2: Suspending Execution Based on a Calculation of Sleep Time

The following example tells SAS to suspend the execution of the DATA step BUDGET until March 1, 2004, at 3:00 AM. SAS calculates the length of the suspension based on the target date and the date and time that the DATA step begins to execute.

```sas
data budget;
    sleeptime='01mar2004:03:00'dt-datetime();
    call sleep(sleeptime,1);
    ...more SAS statements...
run;
```

See Also

Functions:

“SLEEP Function” on page 824

CALL SOFTMAX Routine

Returns the softmax value

Category: Mathematical

Syntax

CALL SOFTMAX(argument<,argument,...>);

Arguments

argument

is numeric.

Restriction: The CALL SOFTMAX routine only accepts variables as valid arguments. Do not use a constant or a SAS expression because the CALL routine is unable to update these arguments.
Details
The CALL SOFTMAX routine replaces each argument with the softmax value of that argument. For example $x_j$ is replaced by

$$e^{x_j} \over \sum_{i=1}^{n} e^{x_i}$$

If any argument contains a missing value, then CALL SOFTMAX returns missing values for all the arguments. Upon a successful return, the sum of all the values is equal to 1.

Examples
The following SAS statements produce these results:

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=0.5;</td>
<td>x=0.3314989604</td>
</tr>
<tr>
<td>y=-0.5;</td>
<td>y=0.1219516523</td>
</tr>
<tr>
<td>z=1;</td>
<td>z=0.5465493873</td>
</tr>
<tr>
<td>call softmax(x,y,z);</td>
<td></td>
</tr>
<tr>
<td>put x y z;</td>
<td></td>
</tr>
</tbody>
</table>

CALL STDIZE Routine

Standardizes the values of one or more variables

Category: Mathematical

Syntax
CALL STDIZE(<option-1, option-2, ...,>variable-1<, variable-2, ...>);

Arguments

option
specifies a character expression whose values can be uppercase, lowercase, or mixed case letters. Leading and trailing blanks are ignored.

Default: STD

Restriction: Use a separate argument for each option because you cannot specify more than one option in a single argument.

Tip: Character expressions can end with an equal sign that is followed by another argument that is a numeric constant, variable, or expression.
See Also: PROC STDIZE in SAS/STAT User’s Guide for information about formulas and other details. The options that are used in CALL STDIZE are the same as those used in PROC STDIZE.

Option includes the following three categories:

**standardization-options**

specify how to compute the location and scale measures that are used to standardize the variables. The following standardization options are available:

**ABW=**

must be followed by an argument that is a numeric expression specifying the tuning constant.

**AGK=**

must be followed by an argument that is a numeric expression that specifies the proportion of pairs to be included in the estimation of the within-cluster variances.

**AHUBER=**

must be followed by an argument that is a numeric expression specifying the tuning constant.

**AWAVE=**

must be followed by an argument that is a numeric expression specifying the tuning constant.

**EUCLLEN**

specifies the Euclidean length.

**IQR**

specifies the interquartile range.

**L=**

must be followed by an argument that is a numeric expression with a value greater than or equal to 1 specifying the power to which differences are to be raised in computing an L(p) or Minkowski metric.

**MAD**

specifies the median absolute deviation from the median.

**MAXABS**

specifies the maximum absolute values.

**MEAN**

specifies the arithmetic mean (average).

**MEDIAN**

specifies the middle number in a set of data that is ordered according to rank.

**MIDRANGE**

specifies the midpoint of the range.

**RANGE**

specifies a range of values.

**SPACING=**

must be followed by an argument that is a numeric expression that specifies the proportion of data to be contained in the spacing.

**STD**

specifies the standard deviation.
SUM
specifies the result that you obtain when you add numbers.

USTD
unstandardizes variables when you also specify the METHOD=IN option.

VARDEF-options
specify the divisor to be used in the calculation of variances. VARDEF options can have the following values:

DF
specifies degrees of freedom.

N
specifies the number of observations.

miscellaneous-options
Miscellaneous options can have the following values:

ADD=
is followed by a numeric argument that specifies a number to add to each value after standardizing and multiplying by the value from the MULT= option. The default value is 0.

FUZZ=
is followed by a numeric argument that specifies the relative fuzz factor.

MISSING=
is followed by a numeric argument that specifies a value to be assigned to variables that have a missing value.

MULT=
is followed by a numeric argument that specifies a number by which to multiply each value after standardizing. The default value is 1.

NORM
normalizes the scale estimator to be consistent for the standard deviation of a normal distribution. This option affects only the methods AGK=, IQR, MAD, and SPACING=.

PSTAT
writes the values of the location and scale measures in the log.

REPLACE
replaces missing values with the value 0 in the standardized data (this value corresponds to the location measure before standardizing). To replace missing values by other values, see the MISSING= option.

SNORM
normalizes the scale estimator to have an expectation of approximately 1 for a standard normal distribution.

Tip: This option affects only the SPACING= method.

Default: DF

variable
is numeric. These values will be standardized according to the method that you use.

Details
The CALL STDIZE routine transforms one or more arguments that are numeric variables by subtracting a location measure and dividing by a scale measure. You can use a variety of location and scale measures.
In addition, you can multiply each standardized value by a constant and you can add a constant. The final output value would be

\[ result = add + mult \times \left( \frac{\text{original} - \text{location}}{\text{scale}} \right) \]

where

- **result** specifies the final value that is returned for each variable.
- **add** specifies the constant to add (ADD= option).
- **mult** specifies the constant to multiply by (MULT= option).
- **original** specifies the original input value.
- **location** specifies the location measure.
- **scale** specifies the scale measure.

You can replace missing values by any constant. If you do not specify the MISSING= or the REPLACE option, variables that have missing values are not altered. The initial estimation method for the ABW=, AHUBER=, and AWAVE= methods is MAD. Percentiles are computed using definition 5. For more information about percentile calculations, see “SAS Elementary Statistics Procedures” in *Base SAS Procedures Guide*.

**Comparisons**

The CALL STDIZE routine is similar to the STDIZE procedure in the SAS/STAT product. However, the CALL STDIZE routine is primarily useful for standardizing the rows of a SAS data set, whereas the STDIZE procedure can standardize only the columns of a SAS data set. For more information, see PROC STDIZE in *SAS/STAT User’s Guide*.

**Examples**

The following SAS statements produce these results.

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>retain x 1 y 2 z 3; call stdize(x,y,z); put x= y= z=;</td>
<td>x=-1 y=0 z=1</td>
</tr>
<tr>
<td>retain w 10 x 11 y 12 z 13; call stdize('iqr',w,x,y,z); put w= x= y= z=;</td>
<td>w=-0.75 x=-0.25 y=0.25 z=0.75</td>
</tr>
<tr>
<td>retain w . x 1 y 2 z 3; call stdize('range',w,x,y,z); put w= x= y= z=;</td>
<td>w=. x=0 y=0.5 z=1</td>
</tr>
<tr>
<td>retain w . x 1 y 2 z 3; call stdize('mult=',10,'missing=', -1,'range',w,x,y,z); put w= x= y= z=;</td>
<td>w=-1 x=0 y=5 z=10</td>
</tr>
</tbody>
</table>
CALL STREAMINIT Routine

Specifies a seed value to use for subsequent random number generation by the RAND function.

**Category:** Random Number

**Syntax**

CALL STREAMINIT(seed);

**Arguments**

*seed* is an integer seed value.

**Range:** seed < \(2^{31} - 1\)

**Tip:** If you specify a nonpositive seed, then CALL STREAMINIT is ignored. Any subsequent random number generation seeds itself from the system clock.

**Details**

If you want to create reproducible streams of random numbers, then specify CALL STREAMINIT before any calls to the RAND random number function. If you call the RAND function before you specify a seed with the CALL STREAMINIT routine (or if you specify a nonpositive seed value in the CALL STREAMINIT routine), then the RAND function uses a call to the system clock to seed itself. For more information about seed values see “Seed Values” on page 257.

**Examples**

**Example 1: Creating a Reproducible Stream of Random Numbers**

The following example shows how to specify a seed value with CALL STREAMINIT to create a reproducible stream of random numbers with the RAND function.

```plaintext
options nodate ps=60 ls=80 pageno=1;

data random;
   call streaminit(123);
   do i=1 to 10;
      x1=rand('cauchy');
      output;
   end;

proc print data=random;
   id i;
run;
```
CALL SYMPUT Routine

Assigns DATA step information to a macro variable

Category: Macro

Syntax

CALL SYMPUT(argument-1,argument-2);

Arguments

argument-1
specifies a character expression that identifies the macro variable that is assigned a value. If the macro variable does not exist, the routine creates it.

argument-2
specifies a character expression that contains the value that is assigned.

Details

The CALL SYMPUT routine either creates a macro variable whose value is information from the DATA step or assigns a DATA step value to an existing macro variable. CALL SYMPUT is fully documented in “SYMPUT Routine” in SAS Macro Language: Reference.
See Also

Function:
“SYMGET Function” on page 845

CALL SYMPUTX Routine

Assigns a value to a macro variable and removes both leading and trailing blanks

Category: Macro

Syntax

CALL SYMPUTX(macro-variable, value <,symbol-table>);

Arguments

macro-variable
can be one of the following:

- a character string that is a SAS name, enclosed in quotation marks.
- the name of a character variable whose values are SAS names.
- a character expression that produces a macro variable name. This form is useful for creating a series of macro variables.

a character constant, variable, or expression. Leading and trailing blanks are removed from the value of name, and the result is then used as the name of the macro variable.

value
specifies a character or numeric constant, variable, or expression. If value is numeric, SAS converts the value to a character string using the BEST. format and does not issue a note to the SAS log. Leading and trailing blanks are removed, and the resulting character string is assigned to the macro variable.

symbol-table
specifies a character constant, variable, or expression. The value of symbol-table is not case sensitive. The first non-blank character in symbol-table specifies the symbol table in which to store the macro variable. The following values are valid as the first non-blank character in symbol-table:

- G specifies that the macro variable is stored in the global symbol table, even if the local symbol table exists.
- L specifies that the macro variable is stored in the most local symbol table that exists, which might be the global symbol table.
- F specifies that if the macro variable exists in any symbol table, CALL SYMPUTX uses the version in the most local symbol table in which it exists. If the macro variable does not exist, CALL SYMPUTX stores the variable in the most local symbol table.
Note: If you omit symbol-table, or if symbol-table is blank, CALL SYMPUTX stores the macro variable in the same symbol table as does the CALL SYMPUT routine.

Comparisons

CALL SYMPUTX is similar to CALL SYMPUT except that

- CALL SYMPUTX does not write a note to the SAS log when the second argument is numeric. CALL SYMPUT, however, writes a note to the log stating that numeric values were converted to character values.
- CALL SYMPUTX uses a field width of up to 32 characters when it converts a numeric second argument to a character value. CALL SYMPUT uses a field width of up to 12 characters.
- CALL SYMPUTX left-justifies both arguments and trims trailing blanks. CALL SYMPUT does not left-justify the arguments, and trims trailing blanks from the first argument only. Leading blanks in the value of name cause an error.
- CALL SYMPUTX enables you to specify the symbol table in which to store the macro variable, whereas CALL SYMPUT does not.

Examples

The following example shows the results of using CALL SYMPUTX.

```sas
data _null_
  call symputx(' items ', ' leading and trailing blanks removed ', 'lplace');
  call symputx(' x ', 123.456);
run;

%put items=!&items!;
%put x=!&x!;
```

The following lines are written to the SAS log:

```
----+----1----+----2----+----3----+----4----+----5
items=!leading and trailing blanks removed!
x=!123.456!
```

See Also

Functions and CALL Routines:

“SYMGET Function” on page 845
“CALL SYMPUT Routine” on page 401

---

CALL SYSTEM Routine

Submits an operating environment command for execution

Category: Special

See: CALL SYSTEM Routine in the documentation for your operating environment.
**CALL TANH Routine**

**Returns** the hyperbolic tangent

**Category:** Mathematical

**Syntax**

CALL TANH(argument<, argument, ...>);

**Arguments**

*argument* is numeric.

**Restriction:** The CALL TANH routine only accepts variables as valid arguments. Do not use a constant or a SAS expression, because the CALL routine is unable to update these arguments.

---

**CALL SYSTEM Routine**

**Syntax**

CALL SYSTEM(command);

**Arguments**

*command* specifies any of the following: a system command that is enclosed in quotation marks (explicit character string), an expression whose value is a system command, or the name of a character variable whose value is a system command that is executed.

**Operating Environment Information:** See the SAS documentation for your operating environment for information on what you can specify.

**Restriction:** The length of the command cannot be greater than 1024 characters, including trailing blanks.

**Comparisons**

The behavior of the CALL SYSTEM routine is similar to that of the X command, the X statement, and the SYSTEM function. It is useful in certain situations because it can be conditionally executed, it accepts an expression as an argument, and it is executed at run time.

**See Also**

Function:

“SYSTEM Function” on page 853
Details

The subroutine TANH replaces each argument by the tanh of that argument. For example $x_j$ is replaced by

$$\tanh(x_j) = \frac{e^{x_j} - e^{-x_j}}{e^{x_j} + e^{-x_j}}$$

If any argument contains a missing value, then CALL TANH returns missing values for all the arguments.

Examples

The following SAS statements produce these results:

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=0.5;</td>
<td></td>
</tr>
<tr>
<td>y=-0.5;</td>
<td></td>
</tr>
<tr>
<td>call tanh(x,y);</td>
<td></td>
</tr>
<tr>
<td>put x= y=;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>x=0.4621171573 y=-0.462117157</td>
</tr>
</tbody>
</table>

See Also

Function:
“TANH Function” on page 855

CALL VNAME Routine

Assigns a variable name as the value of a specified variable

Category: Variable Control

Syntax

CALL VNAME(variable-1,variable-2);

Arguments

variable-1
specifies any SAS variable.

variable-2
specifies any SAS character variable. Because SAS variable names can contain up to 32 characters, the length of variable-2 should be at least 32.
Details
The CALL VNAME routine assigns the name of the variable-1 variable as the value of the variable-2 variable.

Examples
This example uses the CALL VNAME routine with array references to return the names of all variables in the data set OLD:

```sas
data new(keep=name);
set old;
  /* all character variables in old */
array abc{*} _character_
  /* all numeric variables in old */
array def{*} _numeric_
  /* name is not in either array */
length name $32;
do i=1 to dim(abc);
  /* get name of character variable */
call vname(abc{i},name);
  /* write name to an observation */
  output;
end;
do j=1 to dim(def);
  /* get name of numeric variable */
call vname(def{j},name);
  /* write name to an observation */
  output;
end;
stop;
run;
```

See Also
Functions:
“VNAME Function” on page 908
“VNAMEX Function” on page 909

CALL VNEXT Routine

Returns the name, type, and length of a variable that is used in a DATA step

Category: Variable Information

Syntax

CALL VNEXT(varname <,vartype <, varlength>>>);
Arguments

\textit{varname}

is a character variable that is updated by the CALL VNEXT routine. The following rules apply:

- If the input value of \textit{varname} is blank, the value that is returned in \textit{varname} is the name of the first variable in the DATA step's list of variables.
- If the CALL VNEXT routine is executed for the first time in the DATA step, the value that is returned in \textit{varname} is the name of the first variable in the DATA step's list of variables.

If neither of the above conditions exists, the input value of \textit{varname} is ignored. Each time the CALL VNEXT routine is executed, the value that is returned in \textit{varname} is the name of the next variable in the list.

After the names of all the variables in the DATA step are returned, the value that is returned in \textit{varname} is blank.

\textit{vartype}

is a character variable whose input value is ignored. The value that is returned is “N” or “C.” The following rules apply:

- If the value that is returned in \textit{varname} is the name of a numeric variable, the value that is returned in \textit{vartype} is “N.”
- If the value that is returned in \textit{varname} is the name of a character variable, the value that is returned in \textit{vartype} is “C.”
- If the value that is returned in \textit{varname} is blank, the value that is returned in \textit{vartype} is also blank.

\textit{varlength}

is a numeric variable. The input value of \textit{varlength} is ignored.

The value that is returned is the length of the variable whose name is returned in \textit{varname}. If the value that is returned in \textit{varname} is blank, the value that is returned in \textit{varlength} is zero.

Details

The variable names that are returned by the CALL VNEXT routine include automatic variables such as \_N\_ and \_ERROR\_. If the DATA step contains a BY statement, the variable names that are returned by CALL VNEXT include the FIRST.\textit{variable} and LAST.\textit{variable} names. CALL VNEXT also returns the names of the variables that are used as arguments to CALL VNEXT.

\textit{Note:} The order in which variable names are returned by CALL VNEXT can vary in different releases of SAS and in different operating environments.

Examples

The following example shows the results from using the CALL VNEXT routine.

```sas
data test;
  x=1;
  y='abc';
  z=.;
  length z 5;
```

run;

data attributes;
set test;
by x;
input a b $ c;
length name $32 type $3;
name=' '; length=666;
do i=1 to 99 until(name=' ');
call vnext(name,type,length);
put i= name @40 type= length=;
end;
this_is_a_long_variable_name=0;
datalines;
1 q 3
;

The following lines are written to the SAS log:

\[
\begin{align*}
 i\!&=\!1 \quad x & \quad type=N \quad length=8 \\
 i\!&=\!2 \quad y & \quad type=C \quad length=3 \\
 i\!&=\!3 \quad z & \quad type=N \quad length=5 \\
 i\!&=\!4 \quad FIRST.x & \quad type=N \quad length=8 \\
 i\!&=\!5 \quad LAST.x & \quad type=N \quad length=8 \\
 i\!&=\!6 \quad a & \quad type=N \quad length=8 \\
 i\!&=\!7 \quad b & \quad type=C \quad length=8 \\
 i\!&=\!8 \quad c & \quad type=N \quad length=8 \\
 i\!&=\!9 \quad name & \quad type=C \quad length=32 \\
 i\!&=\!10 \quad type & \quad type=C \quad length=3 \\
 i\!&=\!11 \quad length & \quad type=N \quad length=8 \\
 i\!&=\!12 \quad i & \quad type=N \quad length=8 \\
 i\!&=\!13 \quad this_is_a_long_variable_name & \quad type=N \quad length=8 \\
 i\!&=\!14 \quad _ERROR_ & \quad type=N \quad length=8 \\
 i\!&=\!15 \quad _N_ & \quad type=N \quad length=8 \\
 i\!&=\!16 \quad _- & \quad type= \quad length=0 \\
\end{align*}
\]

**CAT Function**

Concatenates character strings without removing leading or trailing blanks

**Category:** Character

**Syntax**

\[
\text{CAT}(\text{string-1} <, \text{... string-n}>)
\]

**Arguments**

\[
\text{string}
\]
specifies a SAS character string.

**Details**

The CAT function returns a value to a variable, or returns a value in a temporary buffer. The value that is returned from the CAT function has a length of up to

- 200 characters in WHERE clauses and in PROC SQL
- 32767 characters in the DATA step except in WHERE clauses
- 65534 characters when `string` is called from the macro processor.

If CAT returns a value in a temporary buffer, the length of the buffer depends on the calling environment, and the value in the buffer can be truncated after CAT finishes processing. In this case, SAS does not write a message about the truncation to the log. If the length of the variable or the buffer is not large enough to contain the result of the concatenation, SAS

- changes the result to a blank value in the DATA step, and in PROC SQL
- writes a warning message to the log stating that the result was either truncated or set to a blank value, depending on the calling environment
- writes a note to the log that shows the location of the function call and lists the argument that caused the truncation
- sets `_ERROR_` to 1 in the DATA step.

The CAT function removes leading and trailing blanks from numeric arguments after it formats the numeric value with the BEST format.

**Comparisons**

The results of the CAT, CATS, CATT, and CATX functions are usually equivalent to those that are produced by certain combinations of the concatenation operator (`||`) and the TRIM and LEFT functions. However, using the CAT, CATS, CATT, and CATX functions is faster than using TRIM and LEFT, and you can use them with the OF syntax for variable lists in calling environments that support variable lists.

The following table shows equivalents of the CAT, CATS, CATT, and CATX functions. The variables X1 through X4 specify character variables, and SP specifies a separator, such as a blank or comma.

<table>
<thead>
<tr>
<th>Function</th>
<th>Equivalent Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAT(OF X1-X4)</td>
<td>X1</td>
</tr>
<tr>
<td>CATS(OF X1-X4)</td>
<td>TRIM(LEFT(X1))</td>
</tr>
<tr>
<td>CATT(OF X1-X4)</td>
<td>TRIM(X1)</td>
</tr>
<tr>
<td>CATX(SP, OF X1-X4)</td>
<td>TRIM(LEFT(X1))</td>
</tr>
</tbody>
</table>

**Examples**

The following example shows how the CAT function concatenates strings.
data _null_;  
x=' The 2002 Olym';  
y='pic Arts Festi';  
z=' val included works by D ‘';  
a='ale Chihuly.’';  
result=cat(x,y,z,a);  
put result $char.;  
run;

The following line is written to the SAS log:

The 2002 Olympic Arts Festival included works by Dale Chihuly.

See Also

Functions and CALL Routines:
  “CATS Function” on page 410
  “CATT Function” on page 412
  “CATX Function” on page 414
  “CALL CATS Routine” on page 337
  “CALL CATT Routine” on page 338
  “CALL CATX Routine” on page 340

CATS Function

Concatenates character strings and removes leading and trailing blanks

Category: Character

Syntax

CATS(string-1 <, ...string-n>)

Arguments

string
    specifies a SAS character string.

Details

The CATS function returns a value to a variable, or returns a value in a temporary buffer. The value that is returned from the CATS function has a length of up to

- 200 characters in WHERE clauses and in PROC SQL
- 32767 characters in the DATA step except in WHERE clauses
- 65534 characters when string is called from the macro processor.
If CATS returns a value in a temporary buffer, the length of the buffer depends on the calling environment, and the value in the buffer can be truncated after CATS finishes processing. In this case, SAS does not write a message about the truncation to the log. If the length of the variable or the buffer is not large enough to contain the result of the concatenation, SAS

- changes the result to a blank value in the DATA step, and in PROC SQL
- writes a warning message to the log stating that the result was either truncated or set to a blank value, depending on the calling environment
- writes a note to the log that shows the location of the function call and lists the argument that caused the truncation
- sets _ERROR_ to 1 in the DATA step.

The CATS function removes leading and trailing blanks from numeric arguments after it formats the numeric value with the BEST format.

Comparisons

The results of the CAT, CATS, CATT, and CATX functions are usually equivalent to those that are produced by certain combinations of the concatenation operator (||) and the TRIM and LEFT functions. However, using the CAT, CATS, CATT, and CATX functions is faster than using TRIM and LEFT, and you can use them with the OF syntax for variable lists in calling environments that support variable lists.

The following table shows equivalents of the CAT, CATS, CATT, and CATX functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Equivalent Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAT(OF X1-X4)</td>
<td>X1</td>
</tr>
<tr>
<td>CATS(OF X1-X4)</td>
<td>TRIM(LEFT(X1))</td>
</tr>
<tr>
<td>CATT(OF X1-X4)</td>
<td>TRIM(X1)</td>
</tr>
<tr>
<td>CATX(SP, OF X1-X4)</td>
<td>TRIM(LEFT(X1))</td>
</tr>
</tbody>
</table>

Examples

The following example shows how the CATS function concatenates strings.

```sas
data _null_
  x=' The Olym'
  y='pic Arts Festi'
  z=' val includes works by D '
  a='ale Chihuly.'
  result=cats(x,y,z,a)
  put result $char.
run;
```
The following line is written to the SAS log:

--+-+--1--+-++--2--+-++--3--+-++--4--+-++--5--+-++--6
The Olympic Arts Festival includes works by Dale Chihuly.

See Also

Functions and CALL Routines:
“CAT Function” on page 408
“CATX Function” on page 414
“CATT Function” on page 412
“CALL CATS Routine” on page 337
“CALL CATT Routine” on page 338
“CALL CATX Routine” on page 340

CATT Function

Concatenates character strings and removes trailing blanks

Category: Character

Syntax

CATT(string-1 <, ...string-n>)

Arguments

string
specifies a SAS character string.

Details

The CATT function returns a value to a variable, or returns a value in a temporary buffer. The value that is returned from the CATT function has a length of up to
- 200 characters in WHERE clauses and in PROC SQL
- 32767 characters in the DATA step except in WHERE clauses
- 65534 characters when string is called from the macro processor.

If CATT returns a value in a temporary buffer, the length of the buffer depends on the calling environment, and the value in the buffer can be truncated after CATT finishes processing. In this case, SAS does not write a message about the truncation to the log. If the length of the variable or the buffer is not large enough to contain the result of the concatenation, SAS
- changes the result to a blank value in the DATA step, and in PROC SQL
- writes a warning message to the log stating that the result was either truncated or set to a blank value, depending on the calling environment
writes a note to the log that shows the location of the function call and lists the argument that caused the truncation

sets _ERROR_ to 1 in the DATA step.

The CATT function removes leading and trailing blanks from numeric arguments after it formats the numeric value with the BEST format.

Comparisons

The results of the CAT, CATS, CATT, and CATX functions are usually equivalent to those that are produced by certain combinations of the concatenation operator (||) and the TRIM and LEFT functions. However, using the CAT, CATS, CATT, and CATX functions is faster than using TRIM and LEFT, and you can use them with the OF syntax for variable lists in calling environments that support variable lists.

The following table shows equivalents of the CAT, CATS, CATT, and CATX functions. The variables X1 through X4 specify character variables, and SP specifies a separator, such as a blank or comma.

<table>
<thead>
<tr>
<th>Function</th>
<th>Equivalent Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAT(OF X1-X4)</td>
<td>X1</td>
</tr>
<tr>
<td>CATS(OF X1-X4)</td>
<td>TRIM(LEFT(X1))</td>
</tr>
<tr>
<td>CATT(OF X1-X4)</td>
<td>TRIM(X1)</td>
</tr>
<tr>
<td>CATX(SP, OF X1-X4)</td>
<td>TRIM(LEFT(X1))</td>
</tr>
</tbody>
</table>

Examples

The following example shows how the CATT function concatenates strings.

data _null_;  
x='The Olym';  
y='pic Arts Festi';  
z='val includes works by Dale Chihuly.';  
a='ale Chihuly.';  
result=catt(x,y,z,a);  
put result $char.;  
run;

The following line is written to the SAS log:

----+----1----+----2----+----3----+----4----+----5----+----6----+----7  
The Olympic Arts Festi val includes works by Dale Chihuly.

See Also

Functions and CALL Routines:
“CAT Function” on page 408  
“CATS Function” on page 410
CATX Function

Concatenates character strings, removes leading and trailing blanks, and inserts separators

Category: Character

Syntax

\[ \text{CATX}(\text{separator, string-1}, ..., \text{string-n}) \]

Arguments

- **separator**: specifies a character string that is used as a separator between concatenated strings.
- **string**: specifies a SAS character string.

Details

The CATX function returns a value to a variable, or returns a value in a temporary buffer. The value that is returned from the CATX function has a length of up to

- 200 characters in WHERE clauses and in PROC SQL
- 32767 characters in the DATA step except in WHERE clauses
- 65534 characters when string is called from the macro processor.

If CATX returns a value in a temporary buffer, the length of the buffer depends on the calling environment, and the value in the buffer can be truncated after CATX finishes processing. In this case, SAS does not write a message about the truncation to the log. If the length of the variable or the buffer is not large enough to contain the result of the concatenation, SAS

- changes the result to a blank value in the DATA step, and in PROC SQL
- writes a warning message to the log stating that the result was either truncated or set to a blank value, depending on the calling environment
- writes a note to the log that shows the location of the function call and lists the argument that caused the truncation
- sets _ERROR_ to 1 in the DATA step.

The CATX function removes leading and trailing blanks from numeric arguments after it formats the numeric value with the BEST. format.

Comparisons

The results of the CAT, CATS, CATT, and CATX functions are usually equivalent to those that are produced by certain combinations of the concatenation operator (|| |) and
the TRIM and LEFT functions. However, using the CAT, CATS, CATT, and CATX functions is faster than using TRIM and LEFT, and you can use them with the OF syntax for variable lists in calling environments that support variable lists.

The following table shows equivalents of the CAT, CATS, CATT, and CATX functions. The variables X1 through X4 specify character variables, and SP specifies a separator, such as a blank or comma.

<table>
<thead>
<tr>
<th>Function</th>
<th>Equivalent Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAT(OF X1-X4)</td>
<td>X1</td>
</tr>
<tr>
<td>CATS(OF X1-X4)</td>
<td>TRIM(LEFT(X1))</td>
</tr>
<tr>
<td>CATT(OF X1-X4)</td>
<td>TRIM(X1)</td>
</tr>
<tr>
<td>CATX(SP, OF X1-X4)</td>
<td>TRIM(LEFT(X1))</td>
</tr>
</tbody>
</table>

### Examples

The following example shows how the CATX function concatenates strings.

```sas
data _null_;  
  separator='%%$%%';  
  x='The Olympic ';  
  y=' Arts Festival ';  
  z=' includes works by ';  
  a='Dale Chihuly.';  
  result=catx(separator,x,y,z,a);  
  put result $char.;  
run;
```

The following line is written to the SAS log:

```
------1-------2-------3-------4-------5-------6-------7
The Olympic%%$%%Arts Festival%%$%%includes works by%%$%%Dale Chihuly.
```

### See Also

Functions and CALL Routines:

- “CAT Function” on page 408
- “CATT Function” on page 412
- “CATS Function” on page 410
- “CALL CATS Routine” on page 337
- “CALL CATT Routine” on page 338
- “CALL CATX Routine” on page 340
CDF Function

Computes cumulative distribution functions

Category: Probability

Syntax

CDF ('dist',quantile,<,parm-1, ... ,parm-k>)

Arguments

'dist'

is a character string that identifies the distribution. Valid distributions are as follows:

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bernoulli</td>
<td>'BERNOULLI'</td>
</tr>
<tr>
<td>Beta</td>
<td>'BETA'</td>
</tr>
<tr>
<td>Binomial</td>
<td>'BINOMIAL'</td>
</tr>
<tr>
<td>Cauchy</td>
<td>'CAUCHY'</td>
</tr>
<tr>
<td>Chi-Square</td>
<td>'CHISQUARE'</td>
</tr>
<tr>
<td>Exponential</td>
<td>'EXPONENTIAL'</td>
</tr>
<tr>
<td>F</td>
<td>'F'</td>
</tr>
<tr>
<td>Gamma</td>
<td>'GAMMA'</td>
</tr>
<tr>
<td>Geometric</td>
<td>'GEOMETRIC'</td>
</tr>
<tr>
<td>Hypergeometric</td>
<td>'HYPERGEOMETRIC'</td>
</tr>
<tr>
<td>Laplace</td>
<td>'LAPLACE'</td>
</tr>
<tr>
<td>Logistic</td>
<td>'LOGISTIC'</td>
</tr>
<tr>
<td>Lognormal</td>
<td>'LOGNORMAL'</td>
</tr>
<tr>
<td>Negative binomial</td>
<td>'NEGBINOMIAL'</td>
</tr>
<tr>
<td>Normal</td>
<td>'NORMAL'</td>
</tr>
<tr>
<td>Normal mixture</td>
<td>'NORMALMIX'</td>
</tr>
<tr>
<td>Pareto</td>
<td>'PARETO'</td>
</tr>
<tr>
<td>Poisson</td>
<td>'POISSON'</td>
</tr>
<tr>
<td>T</td>
<td>'T'</td>
</tr>
<tr>
<td>Uniform</td>
<td>'UNIFORM'</td>
</tr>
<tr>
<td>Wald (inverse Gaussian)</td>
<td>'WALD'</td>
</tr>
<tr>
<td>Weibull</td>
<td>'WEIBULL'</td>
</tr>
</tbody>
</table>
Note: Except for T, F, and NORMALMIX, you can minimally identify any
distribution by its first four characters. △

quantile
is a numeric random variable.

parm-1, ... ,parm-k
are optional shape, location, or scale parameters appropriate for the specific
distribution.

See: “Details” on page 417 for complete information about these parameters

Details
The CDF function computes the left cumulative distribution function from various
continuous and discrete distributions.

Bernoulli Distribution

CDF('BERNOULLI',x,p)

where

x
is a numeric random variable.

p
is a numeric probability of success.

Range: 0 ≤ p ≤ 1

The CDF function for the Bernoulli distribution returns the probability that an
observation from a Bernoulli distribution, with probability of success equal to p, is less
than or equal to x. The equation follows:

\[
CDF('BERN', x, p) = \begin{cases} 
0 & x < 0 \\
1 - p & 0 \leq x < 1 \\
1 & x \geq 1 
\end{cases}
\]

Note: There are no location or scale parameters for this distribution. △

Beta Distribution

CDF('BETA',x,a,b,<l,r>)

where

x
is a numeric random variable.

a
is a numeric shape parameter.

Range: a > 0

b
is a numeric shape parameter.

Range: b > 0

l
is the numeric left location parameter.

Default: 0
\( r \)

is the right location parameter.

**Default:** 1

**Range:** \( r > l \)

The CDF function for the beta distribution returns the probability that an observation from a beta distribution, with shape parameters \( a \) and \( b \), is less than or equal to \( x \). The following equation describes the CDF function of the beta distribution:

\[
CDF(l\text{BETA}, x, a, b, l, r) = \begin{cases} 
0 & x \leq l \\
\frac{1}{\beta(a,b)} \int_{l}^{x} \frac{(x-1)^{a-1}(r-x)^{b-1}}{(r-l)^{a+b-1}} dx & l < x \leq r \\
1 & x > r 
\end{cases}
\]

where

\[
\beta(a, b) = \frac{\Gamma(a) \Gamma(b)}{\Gamma(a + b)}
\]

and

\[
\Gamma(a) = \int_{0}^{\infty} x^{a-1} e^{-x} dx
\]

**Binomial Distribution**

\( \text{CDF('BINOMIAL',} m, p, n) \)

where

\( m \)

is an integer random variable that counts the number of successes.

**Range:** \( m = 0, 1, ... \)

\( p \)

is a numeric probability of success.

**Range:** \( 0 \leq p \leq 1 \)

\( n \)

is an integer parameter that counts the number of independent Bernoulli trials.

**Range:** \( n = 0, 1, ... \)

The CDF function for the binomial distribution returns the probability that an observation from a binomial distribution, with parameters \( p \) and \( n \), is less than or equal to \( m \). The equation follows:

\[
CDF(l\text{BINOM}, m, p, n) = \begin{cases} 
0 & m < 0 \\
\sum_{j=0}^{m} \binom{n}{j} p^j (1 - p)^{n-j} & 0 \leq m \leq n \\
1 & m > n 
\end{cases}
\]
Note: There are no location or scale parameters for the binomial distribution.

Cauchy Distribution

CDF('CAUCHY', x<,θ,λ>)

where

\( x \)

is a numeric random variable.

\( \theta \)

is a numeric location parameter.

Default: 0

\( \lambda \)

is a numeric scale parameter.

Default: 1

Range: \( \lambda > 0 \)

The CDF function for the Cauchy distribution returns the probability that an observation from a Cauchy distribution, with the location parameter \( \theta \) and the scale parameter \( \lambda \), is less than or equal to \( x \). The equation follows:

\[
CDF('CAUCHY', x, \theta, \lambda) = \frac{1}{2} + \frac{1}{\pi} \tan^{-1} \left( \frac{x - \theta}{\lambda} \right)
\]

Chi-Square Distribution

CDF('CHISQUARE', x, df<,nc>)

where

\( x \)

is a numeric random variable.

\( df \)

is a numeric degrees of freedom parameter.

Range: \( df > 0 \)

\( nc \)

is an optional numeric non-centrality parameter.

Range: \( nc \geq 0 \)

The CDF function for the chi-square distribution returns the probability that an observation from a chi-square distribution, with \( df \) degrees of freedom and non-centrality parameter \( nc \), is less than or equal to \( x \). This function accepts non-integer degrees of freedom. If \( nc \) is omitted or equal to zero, the value returned is from the central chi-square distribution. The following equation describes the CDF function of the chi-square distribution:

\[
CDF('CHISQUARE', x, \nu, \lambda) = \begin{cases} 
\sum_{j=0}^{\infty} e^{-\frac{\lambda}{2}} \left( \frac{\lambda}{2} \right)^j j! P_\nu \left( x, \nu + 2j \right) & x < 0 \\
0 & x \geq 0 
\end{cases}
\]
where \( P_{c}(\cdot , \cdot ) \) denotes the probability from the central chi-square distribution:

\[
P_{c}(x, a) = P_{g} \left( \frac{x}{2} ; \frac{a}{2} \right)
\]

and where \( P_{g}(y,b) \) is the probability from the Gamma distribution given by

\[
P_{g}(y, b) = \frac{y^{b-1}}{\Gamma(b)} \int_{0}^{y} e^{-v} v^{b-1} dv
\]

**Exponential Distribution**

\[
\text{CDF}('\text{EXPONENTIAL}', x <',\lambda>')
\]

where

- \( x \)
  - is a numeric random variable.
- \( \lambda \)
  - is a scale parameter.
- **Default:** 1
- **Range:** \( \lambda > 0 \)

The CDF function for the exponential distribution returns the probability that an observation from an exponential distribution, with the scale parameter \( \lambda \), is less than or equal to \( x \). The equation follows:

\[
\text{CDF}('\text{EXPONENTIAL}', x, \lambda) = \begin{cases} 
0 & x < 0 \\
1 - e^{-x} & x \geq 0 
\end{cases}
\]

**F Distribution**

\[
\text{CDF('F', x, ndf, ddf <',nc>)}
\]

where

- \( x \)
  - is a numeric random variable.
- \( \text{ndf} \)
  - is a numeric numerator degrees of freedom parameter.
  - **Range:** \( \text{ndf} > 0 \)
- \( \text{ddf} \)
  - is a numeric denominator degrees of freedom parameter.
  - **Range:** \( \text{ddf} > 0 \)
- \( \text{nc} \)
  - is a numeric non-centrality parameter.
  - **Range:** \( \text{nc} \geq 0 \)

The CDF function for the F distribution returns the probability that an observation from an F distribution, with \( \text{ndf} \) numerator degrees of freedom, \( \text{ddf} \) denominator degrees of freedom, and non-centrality parameter \( \text{nc} \), is less than or equal to \( x \). This
function accepts non-integer degrees of freedom for \( ndf \) and \( ddf \). If \( nc \) is omitted or equal to zero, the value returned is from a central \( F \) distribution. The following equation describes the CDF function of the \( F \) distribution:

\[
CDF \left( ^{t}F^{t}, x, v_1, v_2, \lambda \right) = \begin{cases} 0 & x < 0 \\ \sum_{j=0}^{\infty} e^{-\frac{x}{2}} \frac{\left(\frac{v_2}{2}\right)^j}{\frac{j! (\frac{v_1}{2})^{\frac{j}{2}}}} P_{f}(f, v_1 + 2j, v_2) & x \geq 0 \end{cases}
\]

where \( P_{f}(f, u_1, u_2) \) is the probability from the central \( F \) distribution with

\[
P_{f}(f, u_1, u_2) = P_{B} \left( \frac{u_1 f}{u_1 f + u_2}, \frac{u_1}{2}, \frac{u_2}{2} \right)
\]

and \( P_{b}(x, a, b) \) is the probability from the standard beta distribution.

Note: There are no location or scale parameters for the \( F \) distribution.

**Gamma Distribution**

\[CDF(\text{`GAMMA'}, x, a, \lambda)\]

where

\[x\]

is a numeric random variable.

\[a\]

is a numeric shape parameter.

**Range:** \( a > 0 \)

\[\lambda\]

is a numeric scale parameter.

**Default:** 1

**Range:** \( \lambda > 0 \)

The CDF function for the gamma distribution returns the probability that an observation from a gamma distribution, with shape parameter \( a \) and scale parameter \( \lambda \), is less than or equal to \( x \). The equation follows:

\[
CDF \left( ^{t}GAMMA^{t}, x, a, \lambda \right) = \begin{cases} 0 & x < 0 \\ \frac{1}{\lambda \Gamma(a)} \int_{0}^{x} v^{a-1} e^{-\frac{v}{\lambda}} dv & x \geq 0 \end{cases}
\]

**Geometric Distribution**

\[CDF(\text{`GEOMETRIC'}, m, p)\]

where

\[m\]

is a numeric random variable that denotes the number of failures.

**Range:** \( m = 0, 1, ... \)
is a numeric probability of success.

**Range:** $0 \leq p \leq 1$

The CDF function for the geometric distribution returns the probability that an observation from a geometric distribution, with parameter $p$, is less than or equal to $m$. The equation follows:

$$CDF\left(GEOM^t, m, p\right) = \left\{ \begin{array}{ll}
0 & m < 0 \\
\sum_{j=0}^{m} p(1 - p)^j & m \geq 0
\end{array} \right.$$

**Note:** There are no location or scale parameters for this distribution.

**Hypergeometric Distribution**

CDF('HYPER', x, N, R, n, o, <>)

where

- $x$ is an integer random variable.
- $N$ is an integer population size parameter.
  **Range:** $N = 1, 2, ...$
- $R$ is an integer number of items in the category of interest.
  **Range:** $R = 0, 1, ..., N$
- $n$ is an integer sample size parameter.
  **Range:** $n = 1, 2, ..., N$
- $o$ is an optional numeric odds ratio parameter.
  **Range:** $o > 0$

The CDF function for the hypergeometric distribution returns the probability that an observation from an extended hypergeometric distribution, with population size $N$, number of items $R$, sample size $n$, and odds ratio $o$, is less than or equal to $x$. If $o$ is omitted or equal to 1, the value returned is from the usual hypergeometric distribution. The equation follows:

$$CDF\left(HYPER^t, x, N, R, n, o\right) = \left\{ \begin{array}{ll}
0 & x < \max(0, R + n - N) \\
\sum_{i=0}^{\min \left( R, \frac{N - R}{n - i} \cdot o^i \right)} \left( \begin{array}{c} R \\ i \end{array} \right) \left( \begin{array}{c} N - R \\ n - i \end{array} \right) \left( \begin{array}{c} R \\ j \end{array} \right) \left( \begin{array}{c} N - R \\ n - j \end{array} \right) o^i & \max(0, R + n - N) \leq x \leq \min(R, n) \\
1 & x > \min(R, n)
\end{array} \right.$$

**Laplace Distribution**

CDF('LAPLACE', x < y1, y2)
Functions and CALL Routines

CDF Function

where

- $x$ is a numeric random variable.
- $\theta$ is a numeric location parameter.
- $\lambda$ is a numeric scale parameter.

**Default:**
- $\theta = 0$
- $\lambda = 1$

**Range:** $\lambda > 0$

The CDF function for the Laplace distribution returns the probability that an observation from the Laplace distribution, with the location parameter $\theta$ and the scale parameter $\lambda$, is less than or equal to $x$. The equation follows:

$$CDF \left( 'LAPLACE' \right, x, \theta, \lambda \right) = \begin{cases} \frac{1}{2} e^{\frac{x-\theta}{\lambda}} & x > 0 \\ 1 - \frac{1}{2} e^{\frac{x-\theta}{\lambda}} & x \geq 0 \end{cases}$$

**Logistic Distribution**

CDF('LOGISTIC', x<\theta,\lambda>)

where
- $x$ is a numeric random variable.
- $\theta$ is a numeric location parameter
  - **Default:** $\theta = 0$
- $\lambda$ is a numeric scale parameter.
  - **Default:** $\lambda = 1$
  - **Range:** $\lambda > 0$

The CDF function for the logistic distribution returns the probability that an observation from a logistic distribution, with a location parameter $\theta$ and a scale parameter $\lambda$, is less than or equal to $x$. The equation follows:

$$CDF \left( 'LOGISTIC' \right, x, \theta, \lambda \right) = \frac{1}{1 + e^{\frac{-x-\theta}{\lambda}}}$$

**Lognormal Distribution**

CDF('LOGNORMAL', x<\theta,\lambda>)

where
- $x$ is a numeric random variable.
- $\theta$ is a numeric location parameter.
CDF Function

Chapter 4

Default: 0

\( \lambda \)

is a numeric scale parameter.

Default: 1

Range: \( \lambda > 0 \)

The CDF function for the lognormal distribution returns the probability that an observation from a lognormal distribution, with the location parameter \( \theta \) and the scale parameter \( \lambda \), is less than or equal to \( x \). The equation follows:

\[
CDF('LOGN', x, \theta, \lambda) = \begin{cases} 
0 & x \leq 0 \\
\frac{1}{\lambda \sqrt{2\pi}} \int_{0}^{\log(x)} e^{-(v-\theta)^2/2\lambda^2} dv & x > 0
\end{cases}
\]

Negative Binomial Distribution

CDF('NEGBINOMIAL', \(m\), \(p\), \(n\))

where

\( m \)

is a positive integer random variable that counts the number of failures.

Range: \( m = 0, 1, ... \)

\( p \)

is a numeric probability of success.

Range: \( 0 \leq p \leq 1 \)

\( n \)

is an integer parameter that counts the number of successes.

Range: \( n = 1, 2, ... \)

The CDF function for the negative binomial distribution returns the probability that an observation from a negative binomial distribution, with probability of success \( p \) and number of successes \( n \), is less than or equal to \( m \). The equation follows:

\[
CDF('NEGBINOMIAL', m, p, n) = \begin{cases} 
0 & m < 0 \\
\sum_{j=0}^{m} \binom{n+j-1}{j} (1-p)^j p^n & m \geq 0
\end{cases}
\]

Note: There are no location or scale parameters for the negative binomial distribution.

Normal Distribution

CDF('NORMAL', \(x<\theta, \lambda>\))

where

\( x \)

is a numeric random variable.

\( \theta \)

is a numeric location parameter.

Default: 0
$\lambda$

is a numeric scale parameter.

**Default:** 1

**Range:** $\lambda > 0$

The CDF function for the normal distribution returns the probability that an observation from the normal distribution, with the location parameter $\theta$ and the scale parameter $\lambda$, is less than or equal to $x$. The equation follows:

$$CDF \left( ^{t}NORMAL', x, \theta, \lambda \right) = \frac{1}{\lambda \sqrt{2\pi}} \int_{-\infty}^{x} e^{exp \left( - \frac{(v - \theta)^2}{2\lambda^2} \right)} dv$$

**Normal Mixture Distribution**

**CDF('NORMALMIX',x,n,p,m,s)**

where

$x$

is a numeric random variable.

$n$

is the integer number of mixtures.

**Range:** $n = 1, 2, ...$

$p$

is the $n$ proportions, $p_1, p_2, \ldots, p_n$, where $\sum_{i=1}^{i=n} p_i = 1$.

**Range:** $p = 0, 1, ...$

$m$

is the $n$ means $m_1, m_2, \ldots, m_n$.

$s$

is the $n$ standard deviations $s_1, s_2, \ldots, s_n$.

**Range:** $s > 0$

The CDF function for the normal mixture distribution returns the probability that an observation from a mixture of normal distribution is less than or equal to $x$. The equation follows:

$$CDF \left( ^{t}NORMALMIX', x, n, p, m, s \right) = \sum_{i=1}^{i=n} p_i \ CDF \left( ^{t}NORMAL', x, m_i, s_i \right)$$

**Note:** There are no location or scale parameters for the normal mixture distribution.

**Pareto Distribution**

**CDF('PARETO',x,a,<,k>)**

where

$x$

is a numeric random variable.
$a$
is a numeric shape parameter.

**Range:** $a > 0$

$k$
is a numeric scale parameter.

**Default:** 1

**Range:** $k > 0$

The CDF function for the Pareto distribution returns the probability that an observation from a Pareto distribution, with the shape parameter $a$ and the scale parameter $k$, is less than or equal to $x$. The equation follows:

$$CDF(PARETO', x, a, k) = \begin{cases} 0 & x < k \\ 1 - \left(\frac{k}{x}\right)^a & x \geq k \end{cases}$$

**Poisson Distribution**

$CDF(POISSON', n, m)$

where

$n$
is an integer random variable.

**Range:** $n = 0, 1, ...$

$m$
is a numeric mean parameter.

**Range:** $m > 0$

The CDF function for the Poisson distribution returns the probability that an observation from a Poisson distribution, with mean $m$, is less than or equal to $n$. The equation follows:

$$CDF(POISSON', n, m) = \begin{cases} 0 & n < 0 \\ \sum_{i=0}^{n} \frac{e^{-m} m^i}{i!} & n \geq 0 \end{cases}$$

**Note:** There are no location or scale parameters for the Poisson distribution.

**T Distribution**

$CDF(T', t, df<, nc>)$

where

$t$
is a numeric random variable.

$df$
is a numeric degrees of freedom parameter.

**Range:** $df > 0$

$nc$
is an optional numeric non-centrality parameter.

The CDF function for the $T$ distribution returns the probability that an observation from a $T$ distribution, with degrees of freedom $df$ and non-centrality parameter $nc$, is
less than or equal to \( x \). This function accepts non-integer degrees of freedom. If \( nc \) is omitted or equal to zero, the value returned is from the central \( T \) distribution. The equation follows:

\[
CDF\left(\frac{t}{T}, t, v, \delta \right) = \frac{1}{2\left(\frac{v}{2}\right)\Gamma\left(\frac{v}{2}\right)} \int_{0}^{\infty} x^{v-1} e^{-\frac{1}{2}x^2} \cdot \frac{1}{\sqrt{2\pi}} \int_{\infty}^{\frac{x}{\sqrt{2}}} e^{-\frac{1}{2}(u-\delta)^2} du dx
\]

**Note:** There are no location or scale parameters for the \( T \) distribution.

### Uniform Distribution

\[
CDF\text{('UNIFORM',} x <, l, r >)\]

where

- \( x \) is a numeric random variable.
- \( l \) is the numeric left location parameter.
  - **Default:** 0
- \( r \) is the numeric right location parameter.
  - **Default:** 1
  - **Range:** \( r > l \)

The CDF function for the uniform distribution returns the probability that an observation from a uniform distribution, with the left location parameter \( l \) and the right location parameter \( r \), is less than or equal to \( x \). The equation follows:

\[
CDF\text{('UNIFORM',} x, l, r >) = \begin{cases} 
0 & x < l \\
\frac{x-l}{r-l} & l \leq x < r \\
1 & x \geq r 
\end{cases}
\]

**Note:** The default values for \( l \) and \( r \) are 0 and 1, respectively.

### Wald (Inverse Gaussian) Distribution

\[
CDF\text{('WALD',} x, d)\]

\[
CDF\text{('IGAUSS',} x, d)\]

where

- \( x \) is a numeric random variable.
- \( d \) is a numeric shape parameter.
  - **Range:** \( d > 0 \)
The CDF function for the Wald distribution returns the probability that an observation from a Wald distribution, with shape parameter \( d \), is less than or equal to \( x \). The equation follows:

\[
CDF(WA LD', x, d) = \begin{cases} 
0 & x \leq 0 \\
\Phi \left( \left( x - 1 \right) \sqrt{\frac{d}{x}} \right) + e^{2d} \Phi \left( -(x + 1) \sqrt{\frac{d}{x}} \right) & x > 0
\end{cases}
\]

where \( \Phi(.) \) denotes the probability from the standard normal distribution.

**Note:** There are no location or scale parameters for the Wald distribution.

**Weibull Distribution**

\[
CDF('WEIBULL', x, a <, \lambda>)
\]

where

- \( x \)
  - is a numeric random variable.
- \( a \)
  - is a numeric shape parameter.
- **Range:** \( a > 0 \)
- \( \lambda \)
  - is a numeric scale parameter.
- **Default:** 1
- **Range:** \( \lambda > 0 \)

The CDF function for the Weibull distribution returns the probability that an observation from a Weibull distribution, with the shape parameter \( a \) and the scale parameter \( \lambda \) is less than or equal to \( x \). The equation follows:

\[
CDF(WE IBULL', x, a, \lambda) = \begin{cases} 
0 & x < 0 \\
1 - e^{-\left( \frac{x}{\lambda} \right)^a} & x \geq 0
\end{cases}
\]

**Examples**

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>y=cdf('BERN',0,.25);</td>
<td>0.75</td>
</tr>
<tr>
<td>y=cdf('BETA',0.2,3,4);</td>
<td>0.09888</td>
</tr>
<tr>
<td>y=cdf('BINOM',4,.5,10);</td>
<td>0.37695</td>
</tr>
<tr>
<td>y=cdf('CAUCHY',2);</td>
<td>0.85242</td>
</tr>
<tr>
<td>y=cdf('CHISQ',11.264,11);</td>
<td>0.57858</td>
</tr>
<tr>
<td>y=cdf('EXPO',1);</td>
<td>0.63212</td>
</tr>
<tr>
<td>y=cdf('F',3.32,2,3);</td>
<td>0.82639</td>
</tr>
<tr>
<td>y=cdf('GAMMA',1,3);</td>
<td>0.080301</td>
</tr>
<tr>
<td>y=cdf('HYPER',2,200,50,10);</td>
<td>0.52367</td>
</tr>
<tr>
<td>y=cdf('LAPLACE',1);</td>
<td>0.81606</td>
</tr>
</tbody>
</table>
### CEIL Function

This function returns the smallest integer that is greater than or equal to the argument, fuzzed to avoid unexpected floating-point results.

**Category:** Truncation

**Syntax**

```
CEIL(argument)
```

**Arguments**

*argument* is numeric.

**Details**

If the argument is within 1E-12 of an integer, the function returns that integer.

**Comparisons**

Unlike the CEILZ function, the CEIL function fuzzes the result. If the argument is within 1E-12 of an integer, the CEIL function fuzzes the result to be equal to that integer. The CEILZ function does not fuzz the result. Therefore, with the CEILZ function you might get unexpected results.

**Examples**

The following SAS statements produce these results.

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>y=cdf('LOGISTIC',1);</code></td>
<td>0.73106</td>
</tr>
<tr>
<td><code>y=cdf('LOGNORMAL',1);</code></td>
<td>0.5</td>
</tr>
<tr>
<td><code>y=cdf('NEGB',1,.5,2);</code></td>
<td>0.5</td>
</tr>
<tr>
<td><code>y=cdf('NORMAL',1.96);</code></td>
<td>0.97500</td>
</tr>
<tr>
<td><code>y=cdf('NORMALMIX',2.3,3,.33,.33,.34,.5,1.5,2.5,.79,1.6,4.3);</code></td>
<td>0.7181</td>
</tr>
<tr>
<td><code>y=cdf('PARETO',1,1);</code></td>
<td>0</td>
</tr>
<tr>
<td><code>y=cdf('POISSON',2,1);</code></td>
<td>0.91970</td>
</tr>
<tr>
<td><code>y=cdf('T',.9,5);</code></td>
<td>0.79531</td>
</tr>
<tr>
<td><code>y=cdf('UNIFORM',0.25);</code></td>
<td>0.25</td>
</tr>
<tr>
<td><code>y=cdf('WALD',1,2);</code></td>
<td>0.62770</td>
</tr>
<tr>
<td><code>y=cdf('WEIBULL',1,2);</code></td>
<td>0.63212</td>
</tr>
</tbody>
</table>
SAS Statements | Results
--- | ---
\texttt{var1=2.1; \ a=ceil(var1); \ put a;} | 3
\texttt{b=ceil(-2.4); \ put b;} | -2
\texttt{c=ceil(1+1.e-11); \ put c;} | 2
\texttt{d=ceil(-1+1.e-11); \ put d;} | 0
\texttt{e=ceil(1+1.e-13); \ put e;} | 1
\texttt{f=ceil(223.456); \ put f;} | 224
\texttt{g=ceil(763); \ put g;} | 763
\texttt{h=ceil(-223.456); \ put h;} | -223

See Also

Functions:

“CEILZ Function” on page 430

CEILZ Function

Returns the smallest integer that is greater than or equal to the argument, using zero fuzzing

Category: Truncation

Syntax

\texttt{CEILZ (argument)}

Arguments

\texttt{argument} is a numeric constant, variable, or expression.

Comparisons

Unlike the CEIL function, the CEILZ function uses zero fuzzing. If the argument is within 1E-12 of an integer, the CEIL function fuzzes the result to be equal to that
integer. The CEILZ function does not fuzz the result. Therefore, with the CEILZ function you might get unexpected results.

## Examples

The following SAS statements produce these results.

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=ceilz(2.1);</td>
<td>3</td>
</tr>
<tr>
<td>put a;</td>
<td></td>
</tr>
<tr>
<td>b=ceilz(-2.4);</td>
<td>-2</td>
</tr>
<tr>
<td>put b;</td>
<td></td>
</tr>
<tr>
<td>c=ceilz(1+1.e-11);</td>
<td>2</td>
</tr>
<tr>
<td>put c;</td>
<td></td>
</tr>
<tr>
<td>d=ceilz(-1+1.e-11);</td>
<td>0</td>
</tr>
<tr>
<td>put d;</td>
<td></td>
</tr>
<tr>
<td>e=ceilz(1+1.e-13);</td>
<td>2</td>
</tr>
<tr>
<td>put e;</td>
<td></td>
</tr>
<tr>
<td>f=ceilz(223.456);</td>
<td>224</td>
</tr>
<tr>
<td>put f;</td>
<td></td>
</tr>
<tr>
<td>g=ceilz(763);</td>
<td>763</td>
</tr>
<tr>
<td>put g;</td>
<td></td>
</tr>
<tr>
<td>h=ceilz(-223.456);</td>
<td>-223</td>
</tr>
<tr>
<td>put h;</td>
<td></td>
</tr>
</tbody>
</table>

## See Also

Functions:
- “CEIL Function” on page 429
- “FLOOR Function” on page 544
- “FLOORZ Function” on page 545
- “INT Function” on page 597
- “INTZ Function” on page 608
- “ROUND Function” on page 790
- “ROUNDE Function” on page 797
- “ROUNDZ Function” on page 798

## CEXIST Function

Verifies the existence of a SAS catalog or SAS catalog entry

**Category:** SAS File I/O
Syntax

CEXIST(entry<,'U'>)

Arguments

entry
specifies a SAS catalog, or the name of an entry in a catalog. If the entry value is a one- or two-level name, then it is assumed to be the name of a catalog. Use a three- or four-level name to test for the existence of an entry within a catalog.

'U'
tests whether the catalog can be opened for updating.

Details

CEXIST returns 1 if the SAS catalog or catalog entry exists, or 0 if the SAS catalog or catalog entry does not exist.

Examples

Example 1: Verifying the Existence of an Entry in a Catalog
This example verifies the existence of the entry X.PROGRAM in LIB.CAT1:

```sas
data _null_
   if cexist("lib.cat1.x.program") then
     put "Entry X.PROGRAM exists";
run;
```

Example 2: Determining if a Catalog Can Be Opened for Update
This example tests whether the catalog LIB.CAT1 exists and can be opened for update. If the catalog does not exist, a message is written to the SAS log. Note that in a macro statement you do not enclose character strings in quotation marks.

```sas
%if %sysfunc(cexist(lib.cat1,u)) %then
  %put The catalog LIB.CAT1 exists and can be opened for update.;
%else
  %put %sysfunc(sysmsg());
```

See Also

Function:

“EXIST Function” on page 513

**CHOOSEC Function**

Returns a character value that represents the results of choosing from a list of arguments

Category: Character
Syntax
CHOOSEC (index-expression, selection-1 <,...,selection-n>)

Arguments

index-expression
specifies a numeric expression.

selection
specifies a character expression that is returned.

Details
The CHOOSEC function uses the value of index-expression to select from the arguments that follow. For example, if index-expression is three, CHOOSEC returns the value of selection-3. If the first argument is negative, the function counts backwards from the list of arguments, and returns that value.

Comparisons
The CHOOSEC function is similar to the CHOSEN function except that CHOOSEC returns a character value while CHOSEN returns a numeric value.

Examples
The following example shows how CHOOSEC chooses from a series of values:

```sas
data _null_;  
  Fruit=choosec(1,'apple','orange','pear','fig');  
  Color=choosec(3,'red','blue','green','yellow');  
  Planet=choosec(2,'Mars','Mercury','Uranus');  
  Sport=choosec(-3,'soccer','baseball','gymnastics','skiing');  
  put Fruit= Color= Planet= Sport=;  
run;
```

SAS writes the following line to the log:

```
Fruit=apple Color=green Planet=Mercury Sport=baseball
```

See Also

Functions:
“CHOSEN Function” on page 433

CHOSEN Function

Returns a numeric value that represents the results of choosing from a list of arguments

Category: Character
Syntax

CHOSEN (index-expression, selection-1 <,...,selection-n>)

Arguments

index-expression
specifies a numeric expression.

selection
specifies a numeric expression that is returned.

Details

The CHOSEN function uses the value of index-expression to select from the arguments that follow. For example, if index-expression is 3, CHOSEN returns the value of selection-3. If the first argument is negative, the function counts backwards from the list of arguments, and returns that value.

Comparisons

The CHOSEN function is similar to the CHOOSEC function except that CHOSEN returns a numeric value while CHOOSEC returns a character value.

Examples

The following example shows how CHOSEN chooses from a series of values:

```sas
data _null_
  ItemNumber=choosen(5,100,50,3784,498,679);
  Rank=choosen(-2,1,2,3,4,5);
  Score=choosen(3,193,627,33,290,5);
  Value=choosen(-5,-37,82985,-991,3,1014,-325,3,54,-618);
  put ItemNumber= Rank= Score= Value=;
run;
```

SAS writes the following line to the log:

```
ItemNumber=679 Rank=4 Score=33 Value=1014
```

See Also

Functions:

“CHOOSEC Function” on page 432

CINV Function

Returns a quantile from the chi-squared distribution
Category: Quantile

Syntax

CINV\((p, df, nc)\)

Arguments

\(p\)

is a numeric probability.

Range: \(0 \leq p < 1\)

\(df\)

is a numeric degrees of freedom parameter.

Range: \(df > 0\)

\(nc\)

is a numeric noncentrality parameter.

Range: \(nc \geq 0\)

Details

The CINV function returns the \(p^{th}\) quantile from the chi-square distribution with degrees of freedom \(df\) and a noncentrality parameter \(nc\). The probability that an observation from a chi-square distribution is less than or equal to the returned quantile is \(p\). This function accepts a noninteger degrees of freedom parameter \(df\).

If the optional parameter \(nc\) is not specified or has the value 0, the quantile from the central chi-square distribution is returned. The noncentrality parameter \(nc\) is defined such that if \(X\) is a normal random variable with mean \(\mu\) and variance 1, \(X^2\) has a noncentral chi-square distribution with \(df=1\) and \(nc = \mu^2\).

\textbf{CAUTION:}\n
For large values of \(nc\), the algorithm could fail; in that case, a missing value is returned.

\textbf{Note:}\n
CINV is the inverse of the PROBCHI function.

Examples

The first statement following shows how to find the 95\(^{th}\) percentile from a central chi-square distribution with 3 degrees of freedom. The second statement shows how to find the 95\(^{th}\) percentile from a noncentral chi-square distribution with 3.5 degrees of freedom and a noncentrality parameter equal to 4.5.

\begin{center}
\begin{tabular}{|l|l|}
\hline
SAS Statements & Results \\
\hline
q1=cinv(.95,3); & 7.8147279033 \\
a2=cinv(.95,3.5,4.5); & 7.504582117 \\
\hline
\end{tabular}
\end{center}
CLOSE Function

Closes a SAS data set

Category: SAS File I/O

Syntax

CLOSE(data-set-id)

Arguments

data-set-id
 specifies the data set identifier that the OPEN function returns.

Details

CLOSE returns 0 if the operation was successful, ≠0 if it was not successful. Close all SAS data sets as soon as they are no longer needed by the application.

Note: All data sets opened within a DATA step are closed automatically at the end of the DATA step.

Examples

□ This example uses OPEN to open the SAS data set PAYROLL. If the data set opens successfully, indicated by a positive value for the variable PAYID, the example uses CLOSE to close the data set.

```sas
%let payid=%sysfunc(open(payroll,is));
%if &payid > 0 %then
  %let rc=%sysfunc(close(&payid));
```

□ This example opens the SAS data set MYDATA within a DATA step. MYDATA is closed automatically at the end of the DATA step. You do not need to use CLOSE to close the file.

```sas
data _null_;  
  dsid=open('mydata','i');  
  if dsid > 0 then do;  
    ...more statements...  
  end;  
run;
```

See Also

Function:

“OPEN Function” on page 690
CNONCT Function

Returns the noncentrality parameter from a chi-squared distribution

Category: Mathematical

Syntax

CNONCT(x,df,prob)

Arguments

\(x\)

is a numeric random variable.

Range: \(x \geq 0\)

\(df\)

is a numeric degrees of freedom parameter.

Range: \(df > 0\)

\(prob\)

is a probability.

Range: \(0 < prob < 1\)

Details

The CNONCT function returns the nonnegative noncentrality parameter from a noncentral chi-square distribution whose parameters are \(x\), \(df\), and \(nc\). If \(prob\) is greater than the probability from the central chi-square distribution with the parameters \(x\) and \(df\), a root to this problem does not exist. In this case a missing value is returned. A Newton-type algorithm is used to find a nonnegative root \(nc\) of the equation

\[ P_c (x|df, nc) = prob \]

where

\[ P_c (x|df, nc) = e^{-\frac{nc}{2}} \sum_{j=0}^{\infty} \left( \frac{nc}{2} \right)^j j! P_g \left( \frac{x}{2} | \frac{df}{2} + j \right) \]

where \(P_g (x|a)\) is the probability from the gamma distribution given by

\[ P_g (x|a) = \frac{1}{\Gamma(a)} \int_0^x t^{a-1} e^{-t} dt \]

If the algorithm fails to converge to a fixed point, a missing value is returned.
Examples

```plaintext
data work;
  x=2;
  df=4;
  do nc=1 to 3 by .5;
    prob=probchi(x,df,nc);
    ncc=cnonct(x,df,prob);
    output;
  end;
run;
proc print;
run;
```

Output 4.18  Computations of the Noncentrality Parameters from the Chi-squared Distribution

<table>
<thead>
<tr>
<th>OBS</th>
<th>x</th>
<th>df</th>
<th>nc</th>
<th>prob</th>
<th>ncc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1.0</td>
<td>0.18611</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>1.5</td>
<td>0.15592</td>
<td>1.5</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
<td>2.0</td>
<td>0.13048</td>
<td>2.0</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2.5</td>
<td>0.10907</td>
<td>2.5</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>4</td>
<td>3.0</td>
<td>0.09109</td>
<td>3.0</td>
</tr>
</tbody>
</table>

**COALESCE Function**

Returns the first non-missing value from a list of numeric arguments.

**Category:** Mathematical

**Syntax**

`COALESCE(argument-1,..., argument-n)`

**Arguments**

- `argument` is numeric. `argument` can be one of the following items:
  - a numeric value
  - a missing numeric value
  - the name of a numeric variable.

**Details**

COALESCE accepts one or more numeric arguments. The COALESCE function checks the value of each argument in the order in which they are listed and returns the first
non-missing value. If only one value is listed, then the COALESCE function returns the value of that argument. If all the values of all arguments are missing, then the COALESCE function returns a missing value.

**Comparisons**
The COALESCE function searches numeric arguments, whereas the COALESCEC function searches character arguments.

**Examples**

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = COALESCE(42, .);</td>
<td>42</td>
</tr>
<tr>
<td>y = COALESCE(.A, .B, .C);</td>
<td>.</td>
</tr>
<tr>
<td>z = COALESCE(., 7, ., ., 42);</td>
<td>7</td>
</tr>
</tbody>
</table>

**See Also**

Function:

“COALESCEC Function” on page 439

**COALESCEC Function**

Returns the first non-missing value from a list of character arguments.

**Category:** Character

**Syntax**

COALESCEC(argument-1<..., argument-n>)

**Arguments**

*argument*

a character value. *argument* can be one of the following items:

- a character value
- a missing character value
- the name of a character variable.
**Details**

COALESCEC accepts one or more character arguments. The COALESCEC function checks the value of each argument in the order in which they are listed and returns the first non-missing value. If only one value is listed, then the COALESCEC function returns the value of that argument. If all the values of all arguments are missing, then the COALESCEC function returns a missing value.

**Comparisons**

The COALESCEC function searches character arguments, whereas the COALESCE function searches numeric arguments.

**Examples**

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>COALESCEC('', 'Hello')</td>
<td>Hello</td>
</tr>
<tr>
<td>COALESCEC('', 'Goodbye', 'Hello')</td>
<td>Goodbye</td>
</tr>
</tbody>
</table>

**See Also**

Function:

“COALESCE Function” on page 438

---

**COLLATE Function**

**Returns an ASCII or EBCDIC collating sequence character string**

**Category:** Character

**See:** COLLATE Function in the documentation for your operating environment.

**Syntax**

COLLATE (start-position<.,end-position>) | (start-position<.,length>)
Arguments

start-position
specifies the numeric position in the collating sequence of the first character to be returned.

Interaction: If you specify only start-position, COLLATE returns consecutive characters from that position to the end of the collating sequence or up to 255 characters, whichever comes first.

end-position
specifies the numeric position in the collating sequence of the last character to be returned.

The maximum end-position for the EBCDIC collating sequence is 255. For ASCII collating sequences, the characters that correspond to end-position values between 0 and 127 represent the standard character set. Other ASCII characters that correspond to end-position values between 128 and 255 are available on certain ASCII operating environments, but the information that those characters represent varies with the operating environment.

Tip: end-position must be larger than start-position
Tip: If you specify end-position, COLLATE returns all character values in the collating sequence between start-position and end-position, inclusive.
Tip: If you omit end-position and use length, mark the end-position place with a comma.

length
specifies the number of characters in the collating sequence.

Default: 200

Tip: If you omit end-position, use length to specify the length of the result explicitly.

Details
If the COLLATE function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

If you specify both end-position and length, COLLATE ignores length. If you request a string longer than the remainder of the sequence, COLLATE returns a string through the end of the sequence.

Examples
The following SAS statements produce these results.

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII</td>
<td></td>
</tr>
<tr>
<td>x=collate(48,,10);</td>
<td>-+-+-+-+</td>
</tr>
<tr>
<td>y=collate(48,57);</td>
<td></td>
</tr>
<tr>
<td>put @1 x @14 y;</td>
<td>0123456789 0123456789</td>
</tr>
<tr>
<td>EBCDIC</td>
<td></td>
</tr>
<tr>
<td>x=collate(240,,10);</td>
<td>-+-+-+-+</td>
</tr>
<tr>
<td>y=collate(240,249);</td>
<td></td>
</tr>
<tr>
<td>put @1 x @14 y;</td>
<td>0123456789 0123456789</td>
</tr>
</tbody>
</table>
See Also

Functions:
“BYTE Function” on page 334
“RANK Function” on page 780

COMB Function

Computes the number of combinations of \( n \) elements taken \( r \) at a time

Category: Mathematical

Syntax
COMB\((n, r)\)

Arguments

\( n \)

is an integer that represents the total number of elements from which the sample is chosen.

\( r \)

is an integer that represents the number of chosen elements.

Restriction: \( r \leq n \)

Details

The mathematical representation of the COMB function is given by the following equation:

\[
COMB(n, r) = \binom{n}{r} = \frac{n!}{r!(n-r)!}
\]

with \( n \geq 0 \), \( r \geq 0 \), and \( n \geq r \).

If the expression cannot be computed, a missing value is returned.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x=\text{comb}(5,1) );</td>
<td>5</td>
</tr>
</tbody>
</table>
COMPARE Function

Returns the position of the leftmost character by which two strings differ, or returns 0 if there is no difference

Category: Character

Syntax

COMPARE(string–1, string–2<,modifiers>)

Arguments

string–1
specifies a character constant, variable, or expression.

string–2
specifies a character constant, variable, or expression.

modifiers
specifies a character string that can modify the action of the COMPARE function.
You can use one or more of the following characters as a valid modifier:

i or I ignores the case in string–1 and string–2.
l or L removes leading blanks in string–1 and string–2 before comparing the values.
n or N removes quotation marks from any argument that is an n-literal and ignores the case of string–1 and string–2.
: (colon) truncates the longer of string–1 or string–2 to the length of the shorter string, or to one, whichever is greater. If you do not specify this modifier, the shorter string is padded with blanks to the same length as the longer string.

TIP: COMPARE ignores blanks that are used as modifiers.

Details

The order in which the modifiers appear in the COMPARE function is relevant.

- “LN” first removes leading blanks from each string, and then removes quotation marks from n-literals.
- “NL” first removes quotation marks from n-literals, and then removes leading blanks from each string.
In the COMPARE function, if string–1 and string–2 do not differ, COMPARE returns a value of zero. If the arguments differ:

- the sign of the result is negative if string–1 precedes string–2 in a sort sequence, and positive if string–1 follows string–2 in a sort sequence
- the magnitude of the result is equal to the position of the leftmost character at which the strings differ.

**Examples**

**Example 1: Understanding the Order of Comparisons When Comparing Two Strings**

The following example compares two strings by using the COMPARE function.

```plaintext
options pageno=1 nodate ls=80 ps=60;

data test;
  infile datalines missover;
  input string1 $char8. string2 $char8. modifiers $char8.;
  result=compare(string1, string2, modifiers);
  datalines;
1234567812345678
123   abc
abc   abx
xyz   abcd
aBc   abc
aBc   AbC  i
   abc  abc
   abc  abc  l
   abc  abx
   abc  abx  l
ABC   'abc'n
ABC   'abc'n  n
'\$12'n $12  n
'\$12'n $12  nl
'\$12'n $12  ln
;

proc print data=test;
run;
```

The following output shows the results.
Output 4.19  Results of Comparing Two Strings by Using the COMPARE Function

<table>
<thead>
<tr>
<th>Obs</th>
<th>string1</th>
<th>string2</th>
<th>modifiers</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12345678</td>
<td>12345678</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>123</td>
<td>abc</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>abc</td>
<td>abx</td>
<td>-3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>xyz</td>
<td>abcd ef</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>ABC</td>
<td>abc</td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>ABC</td>
<td>Abc</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>abc</td>
<td>abc</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>abc</td>
<td>abx</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>abx</td>
<td>abx</td>
<td>-3</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>ABC</td>
<td>'abc'n</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>ABC</td>
<td>'abc'n</td>
<td>n</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>ABC</td>
<td>'abc'n</td>
<td>n</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>'§12'n</td>
<td>$12</td>
<td>n</td>
<td>-1</td>
</tr>
<tr>
<td>14</td>
<td>'§12'n</td>
<td>$12</td>
<td>nl</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>'§12'n</td>
<td>$12</td>
<td>ln</td>
<td>0</td>
</tr>
</tbody>
</table>

Example 2: Truncating Strings Using the COMPARE Function

The following example uses the : (colon) modifier to truncate strings.

```sas
options pageno=1 nodate ls=80 pagesize=60;

data test2;
pad1=compare('abc','abc ');
pad2=compare('abc','abce df ');
truncate1=compare('abc','abcd ef',':');
truncate2=compare('abce df','abc',':');
blank=compare('',' abc',':');
run;

proc print data=test2 noobs;
run;
```

The following output shows the results.

Output 4.20  Results of Using the Truncation Modifier

<table>
<thead>
<tr>
<th>pad1</th>
<th>pad2</th>
<th>truncate1</th>
<th>truncate2</th>
<th>blank</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-4</td>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
</tbody>
</table>

See Also

Functions and CALL Routines:

“COMPGED Function” on page 447
“COMPLEV Function” on page 452
“CALL COMPCOST Routine” on page 342
COMPBL Function

Removes multiple blanks from a character string

Category: Character

Syntax

COMPBL(source)

Arguments

source

specifies the source string to compress.

Details

If the COMPBL function returns a value to a variable that has not yet been assigned a length, by default the variable length is determined by the length of the first argument.

The COMPBL function removes multiple blanks in a character string by translating each occurrence of two or more consecutive blanks into a single blank.

The default length of the value that COMPBL returns is the same as the length of the argument.

Comparisons

The COMPRESS function removes every occurrence of the specific character from a string. If you specify a blank as the character to remove from the source string, the COMPRESS function is similar to the COMPBL function. However, the COMPRESS function removes all blanks from the source string, while the COMPBL function compresses multiple blanks to a single blank and has no affect on a single blank.

Examples

The following SAS statements produce these results.

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>string='Hey Diddle Diddle';</td>
<td>Hey Diddle Diddle</td>
</tr>
<tr>
<td>string=compbl(string);</td>
<td></td>
</tr>
<tr>
<td>put string;</td>
<td></td>
</tr>
<tr>
<td>string='125 E Main St';</td>
<td>125 E Main</td>
</tr>
<tr>
<td>length address $10;</td>
<td></td>
</tr>
<tr>
<td>address=compbl(string);</td>
<td></td>
</tr>
<tr>
<td>put address;</td>
<td></td>
</tr>
</tbody>
</table>
COMPGED Function

Compares two strings by computing the generalized edit distance

Category: Character

Syntax

COMPGED(string-1, string-2 <,cutoff> <,modifiers>)

Arguments

string–1
specifies a character constant, variable, or expression.

string–2
specifies a character constant, variable, or expression.

cutoff
is a numeric constant, variable, or expression. If the actual generalized edit distance
is greater than the value of cutoff, the value that is returned is equal to the value of

cutoff.

Tip: Using a small value of cutoff improves the efficiency of COMPGED if the
values of string–1 and string–2 are long.

modifiers
specifies a character string that can modify the action of the COMPGED function.
You can use one or more of the following characters as a valid modifier:

i or I ignores the case in string–1 and string–2.

l or L removes leading blanks in string–1 and string–2 before comparing
the values.

n or N removes quotation marks from any argument that is an n-literal
and ignores the case of string–1 and string–2.

: (colon) truncates the longer of string–1 or string–2 to the length of the
shorter string, or to one, whichever is greater.

TIP: COMPGED ignores blanks that are used as modifiers.

Details

The Order in Which Modifiers Appear

The order in which the modifiers appear in the
COMPGED function is relevant.
"LN" first removes leading blanks from each string and then removes quotation marks from n-literals.

"NL" first removes quotation marks from n-literals and then removes leading blanks from each string.

**Definition of Generalized Edit Distance** Generalized edit distance is a generalization of Levenshtein edit distance, which is a measure of dissimilarity between two strings. The Levenshtein edit distance is the number of deletions, insertions, or replacements of single characters that are required to transform `string-1` into `string-2`.

**Computing the Generalized Edit Distance** The COMPGED function returns the generalized edit distance between `string-1` and `string-2`. The generalized edit distance is the minimum-cost sequence of operations for constructing `string-1` from `string-2`.

The algorithm for computing the sum of the costs involves a pointer that points to a character in `string-2` (the input string). An output string is constructed by a sequence of operations that might advance the pointer, add one or more characters to the output string, or both. Initially, the pointer points to the first character in the input string, and the output string is empty.

The operations and their costs are described in the following table.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Default Cost in Units</th>
<th>Description of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPEND</td>
<td>50</td>
<td>When the output string is longer than the input string, add any one character to the end of the output string without moving the pointer.</td>
</tr>
<tr>
<td>BLANK</td>
<td>10</td>
<td>Do any of the following:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>□ Add one space character to the end of the output string without moving the pointer.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>□ When the character at the pointer is a space character, advance the pointer by one position without changing the output string.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>□ When the character at the pointer is a space character, add one space character to the end of the output string, and advance the pointer by one position.</td>
</tr>
<tr>
<td>DELETE</td>
<td>100</td>
<td>Advance the pointer by one position without changing the output string.</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>20</td>
<td>Add the character at the pointer to the end of the output string without moving the pointer.</td>
</tr>
<tr>
<td>FDELETE</td>
<td>200</td>
<td>When the output string is empty, advance the pointer by one position without changing the output string.</td>
</tr>
</tbody>
</table>

If the cost for BLANK is set to zero by the COMPCOST function, the COMPGED function removes all space characters from both strings before doing the comparison.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Default Cost in Units</th>
<th>Description of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>FINSERT</td>
<td>200</td>
<td>When the pointer is in position one, add any one character to the end of the output string without moving the pointer.</td>
</tr>
<tr>
<td>FREPLACE</td>
<td>200</td>
<td>When the pointer is in position one and the output string is empty, add any one character to the end of the output string, and advance the pointer by one position.</td>
</tr>
<tr>
<td>INSERT</td>
<td>100</td>
<td>Add any one character to the end of the output string without moving the pointer.</td>
</tr>
<tr>
<td>MATCH</td>
<td>0</td>
<td>Copy the character at the pointer from the input string to the end of the output string, and advance the pointer by one position.</td>
</tr>
<tr>
<td>PUNCTUATION</td>
<td>30</td>
<td>Do any of the following:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>□ Add one punctuation character to the end of the output string without moving the pointer.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>□ When the character at the pointer is a punctuation character, advance the pointer by one position without changing the output string.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>□ When the character at the pointer is a punctuation character, add one punctuation character to the end of the output string, and advance the pointer by one position.</td>
</tr>
<tr>
<td>REPLACE</td>
<td>100</td>
<td>Add any one character to the end of the output string, and advance the pointer by one position.</td>
</tr>
<tr>
<td>SINGLE</td>
<td>20</td>
<td>When the character at the pointer is the same as the character that follows in the input string, advance the pointer by one position without changing the output string.</td>
</tr>
</tbody>
</table>

If the cost for PUNCTUATION is set to zero by the COMPCOST function, the COMPGED function removes all punctuation characters from both strings before doing the comparison.
To set the cost of the string operations, you can use the CALL COMPCOST routine or use default costs. If you use the default costs, the values that are returned by COMPGED are approximately 100 times greater than the values that are returned by COMPLEV.

**Examples of Errors**  The rationale for determining the generalized edit distance is based on the number and kinds of typographical errors that can occur. COMPGED assigns a cost to each error and determines the minimum sum of these costs that could be incurred. Some kinds of errors can be more serious than others. For example, inserting an extra letter at the beginning of a string might be more serious than omitting a letter from the end of a string. For another example, if you type a word or phrase that exists in `string-2` and introduce a typographical error, you might produce `string-1` instead of `string-2`.

**Making the Generalized Edit Distance Symmetric**  Generalized edit distance is not necessarily symmetric. That is, the value that is returned by `COMPGED(string1, string2)` is not always equal to the value that is returned by `COMPGED(string2, string1)`. To make the generalized edit distance symmetric, use the CALL COMPCOST routine to assign equal costs to the operations within each of the following pairs:

- INSERT, DELETE
- FINSERT, FDELETE
- APPEND, TRUNCATE
- DOUBLE, SINGLE

**Comparisons**

You can compute the Levenshtein edit distance by using the COMPLEV function. You can compute the generalized edit distance by using the CALL COMPCOST routine and the COMPGED function. Computing generalized edit distance requires considerably more computer time than does computing Levenshtein edit distance. But generalized edit distance usually provides a more useful measure than Levenshtein edit distance for applications such as fuzzy file merging and text mining.

**Examples**

The following example uses the default costs to calculate the generalized edit distance.

```plaintext
options nodate pageno=1 linesize=70 pagesize=60;
```
data test;
  infile datalines missover;
  input String1 $char8. +1 String2 $char8. +1 Operation $40.;
  GED=compged(string1, string2);
  datalines;
  baboon  baboon  match
  baXboon baboon  insert
  baoon  baboon  delete
  baXoon baboon  replace
  baboonX baboon  append
  baboo  baboon  truncate
  babboon baboon  double
  babon  baboon  single
  baobon baboon  swap
  bab oon baboon  blank
  bab,oon baboon  punctuation
  bXaoon  baboon  insert+delete
  bXaYoon baboon  insert+replace
  bXoon  baboon  delete+replace
  Xbaboon baboon  finsert
  aboon  baboon  trick question: swap+delete
  Xaboon baboon  freplace
  axoon  baboon  fdelete+replace
  axoo  baboon  fdelete+replace+truncate
  axon  baboon  fdelete+replace+single
  baby  baboon  replace+truncate*2
  balloon baboon  replace+insert
;
  proc print data=test label;
    label GED='Generalized Edit Distance';
    var String1 String2 GED Operation;
  run;
  The following output shows the results.
The SAS System

Generalized Edit Distance Based on Operation

<table>
<thead>
<tr>
<th>Obs</th>
<th>String1</th>
<th>String2</th>
<th>Distance</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>baboon</td>
<td>baboon</td>
<td>0</td>
<td>match</td>
</tr>
<tr>
<td>2</td>
<td>baXboon</td>
<td>baboon</td>
<td>100</td>
<td>insert</td>
</tr>
<tr>
<td>3</td>
<td>baoon</td>
<td>baboon</td>
<td>100</td>
<td>delete</td>
</tr>
<tr>
<td>4</td>
<td>baXoon</td>
<td>baboon</td>
<td>100</td>
<td>replace</td>
</tr>
<tr>
<td>5</td>
<td>baboonX</td>
<td>baboon</td>
<td>50</td>
<td>append</td>
</tr>
<tr>
<td>6</td>
<td>baboo</td>
<td>baboon</td>
<td>10</td>
<td>truncate</td>
</tr>
<tr>
<td>7</td>
<td>babboon</td>
<td>baboon</td>
<td>20</td>
<td>double</td>
</tr>
<tr>
<td>8</td>
<td>babon</td>
<td>baboon</td>
<td>20</td>
<td>single</td>
</tr>
<tr>
<td>9</td>
<td>babon</td>
<td>baboon</td>
<td>20</td>
<td>swap</td>
</tr>
<tr>
<td>10</td>
<td>bab oon</td>
<td>baboon</td>
<td>10</td>
<td>blank</td>
</tr>
<tr>
<td>11</td>
<td>bab oon</td>
<td>baboon</td>
<td>30</td>
<td>punctuation</td>
</tr>
<tr>
<td>12</td>
<td>bXaon</td>
<td>baboon</td>
<td>200</td>
<td>insert+delete</td>
</tr>
<tr>
<td>13</td>
<td>bXaYoon</td>
<td>baboon</td>
<td>200</td>
<td>insert+replace</td>
</tr>
<tr>
<td>14</td>
<td>bXoon</td>
<td>baboon</td>
<td>200</td>
<td>delete+replace</td>
</tr>
<tr>
<td>15</td>
<td>Xbabo</td>
<td>baboon</td>
<td>200</td>
<td>finsert</td>
</tr>
<tr>
<td>16</td>
<td>aboon</td>
<td>baboon</td>
<td>200</td>
<td>trick question: swap+delete</td>
</tr>
<tr>
<td>17</td>
<td>Xaboon</td>
<td>baboon</td>
<td>200</td>
<td>freplace</td>
</tr>
<tr>
<td>18</td>
<td>axoon</td>
<td>baboon</td>
<td>300</td>
<td>fdelete+replace</td>
</tr>
<tr>
<td>19</td>
<td>axoo</td>
<td>baboon</td>
<td>310</td>
<td>fdelete+replace+truncate</td>
</tr>
<tr>
<td>20</td>
<td>axon</td>
<td>baboon</td>
<td>320</td>
<td>fdelete+replace+single</td>
</tr>
<tr>
<td>21</td>
<td>baby</td>
<td>baboon</td>
<td>120</td>
<td>replace+truncate*2</td>
</tr>
<tr>
<td>22</td>
<td>balloon</td>
<td>baboon</td>
<td>200</td>
<td>replace+insert</td>
</tr>
</tbody>
</table>

See Also

Functions:
“COMPARE Function” on page 443
“CALL COMPCOST Routine” on page 342
“COMPLEV Function” on page 452

COMPLEV Function

Compares two strings by computing the Levenshtein edit distance

Category: Character

Syntax

COMPLEV(string-1, string-2 <,cutoff> <,modifiers>)

Arguments

string-1
specifies a character constant, variable, or expression.
string–2
specifies a character constant, variable, or expression.

cutoff
specifies a numeric constant, variable, or expression. If the actual Levenshtein edit
distance is greater than the value of cutoff, the value that is returned is equal to the
value of cutoff.
**Tip:** Using a small value of cutoff improves the efficiency of COMPGED if the
values of string–1 and string–2 are long.

modifiers
specifies a character string that can modify the action of the COMPLEV function.
You can use one or more of the following characters as a valid modifier:
i or I ignores the case in string–1 and string–2.
l or L removes leading blanks in string–1 and string–2 before comparing
the values.
n or N removes quotation marks from any argument that is an n-literal
and ignores the case of string–1 and string–2.
: (colon) truncates the longer of string–1 or string–2 to the length of the
shorter string, or to one, whichever is greater.
**TIP:** COMPLEV ignores blanks that are used as modifiers.

Details
The order in which the modifiers appear in the COMPLEV function is relevant.
- “LN” first removes leading blanks from each string and then removes quotation
  marks from n-literals.
- “NL” first removes quotation marks from n-literals and then removes leading
  blanks from each string.

The COMPLEV function ignores trailing blanks.
COMPLEV returns the Levenshtein edit distance between string–1 and string–2.
Levenshtein edit distance is the number of insertions, deletions, or replacements of
single characters that are required to convert one string to the other. Levenshtein edit
distance is symmetric. That is, COMPLEV(string–1,string–2) is the same as
COMPLEV(string–2,string–1).

Comparisons
The Levenshtein edit distance that is computed by COMPLEV is a special case of the
generalized edit distance that is computed by COMPGED.
COMPLEV executes much more quickly than COMPGED.

Examples
The following example compares two strings by computing the Levenshtein edit
distance.

options pageno=1 nodate ls=80 ps=60;

data test;
ingfile datalines missover;
input string1 $char8. string2 $char8. modifiers $char8.;
result=complev(string1, string2, modifiers);
datalines;
1234567812345678
abc  abxc
ac   abc
aXc  abc
aXbZc abc
aXYZc abc
WaXbYcZ abc
XYZ  abcdef
aBc  abc
ABc  AbC i
abc  abc
abc  abc 1
AXC  'abc'n
AXC  'abc'n n
;

proc print data=test;
run;

The following output shows the results.

Output 4.22  Results of Comparing Two Strings by Computing the Levenshtein Edit Distance

<table>
<thead>
<tr>
<th>Obs</th>
<th>string1</th>
<th>string2</th>
<th>modifiers</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12345678</td>
<td>12345678</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>abc</td>
<td>abxc</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>ac</td>
<td>abc</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>aXc</td>
<td>abc</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>aXbZc</td>
<td>abc</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>aXYZc</td>
<td>abc</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>WaXbYcZ</td>
<td>abc</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>XYZ</td>
<td>abcdef</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>aBc</td>
<td>abc</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>aBc</td>
<td>AbC</td>
<td>i 0</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>abc</td>
<td>abc</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>abc</td>
<td>abc</td>
<td>1 0</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>AXC</td>
<td>'abc'n</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>AXC</td>
<td>'abc'n</td>
<td>n 1</td>
<td></td>
</tr>
</tbody>
</table>

See Also

Functions and CALL Routines:

“COMPARE Function” on page 443
“COMPGED Function” on page 447
“CALL COMPCOST Routine” on page 342
**COMPOUND Function**

Returns compound interest parameters

Category: Financial

**Syntax**

```plaintext
COMPOUND(a,f,r,n)
```

**Arguments**

- `a`
  - is numeric, the initial amount.
  - **Range:** `a ≥ 0`

- `f`
  - is numeric, the future amount (at the end of `n` periods).
  - **Range:** `f ≥ 0`

- `r`
  - is numeric, the periodic interest rate expressed as a fraction.
  - **Range:** `r ≥ 0`

- `n`
  - is an integer, the number of compounding periods.
  - **Range:** `n ≥ 0`

**Details**

The COMPOUND function returns the missing argument in the list of four arguments from a compound interest calculation. The arguments are related by

\[
f = a \left(1 + r\right)^n
\]

One missing argument must be provided. It is then calculated from the remaining three. No adjustment is made to convert the results to round numbers.

**Examples**

The accumulated value of an investment of $2000 at a nominal annual interest rate of 9 percent, compounded monthly after 30 months, can be expressed as

```plaintext
future=compound(2000,.,0.09/12,30);
```

The value returned is 2502.54. The second argument has been set to missing, indicating that the future amount is to be calculated. The 9 percent nominal annual rate has been converted to a monthly rate of 0.09/12. The rate argument is the fractional (not the percentage) interest rate per compounding period.
COMPRESS Function

Removes specific characters from a character string

Category: Character

Syntax

COMPRESS(<source>, chars, modifiers)

Arguments

source
specifies a source string that contains characters to remove.

chars
specifies a character string that initializes a list of characters. By default, the characters in this list are removed from the source. If you specify the “K” modifier in the third argument, then only the characters in this list are kept in the result.

Tip: You can add more characters to this list by using other modifiers in the third argument.

Tip: Enclose a literal string of characters in quotation marks.

modifiers
specifies a character string in which each character modifies the action of the COMPRESS function. Blanks are ignored. These are the characters that can be used as modifiers:

a or A       adds letters of the Latin alphabet (A - Z, a - z) to the list of characters.
c or C       adds control characters to the list of characters.
d or D       adds numerals to the list of characters.
f or F       adds the underscore character and letters of the Latin alphabet (A - Z, a - z) to the list of characters.
g or G       adds graphic characters to the list of characters.
i or I       ignores the case of the characters to be kept or removed.
k or K       keeps the characters in the list instead of removing them.
l or L       adds lowercase letters (a - z) to the list of characters.
n or N       adds numerals, the underscore character, and letters of the Latin alphabet (A - Z, a - z) to the list of characters.
o or O       processes the second and third arguments once rather than every time the COMPRESS function is called. Using the “O” modifier in the DATA step (excluding WHERE clauses) or the SQL procedure can make COMPRESS run much faster when you call it in a loop where the second and third arguments do not change.
p or P       adds punctuation marks to the list of characters.
s or S  adds space characters to the list of characters (blank, horizontal tab, vertical tab, carriage return, line feed, and form feed).

t or T  trims trailing blanks from the first and second arguments.

u or U  adds uppercase letters (A - Z) to the list of characters.

w or W  adds printable characters to the list of characters ("W" for writable, because "P" is used for punctuation).

x or X  adds hexadecimal characters to the list of characters.

**Details**

The COMPRESS function allows null arguments. A null argument is treated as a string that has a length of zero.

Based on the number of arguments, the COMPRESS functions works as follows:

<table>
<thead>
<tr>
<th>If you call the COMPRESS function with...</th>
<th>The result is...</th>
</tr>
</thead>
<tbody>
<tr>
<td>only the first argument, source</td>
<td>The argument with all blanks removed. If the argument is completely blank, then the result is a string with a length of zero. If you assign the result to a character variable with a fixed length, then the value of that variable will be padded with blanks to fill its defined length.</td>
</tr>
<tr>
<td>the first two arguments, source and chars</td>
<td>All characters that appear in the second argument are removed from the result.</td>
</tr>
<tr>
<td>three arguments, source, chars, and modifiers</td>
<td>The “k” or “K” modifier (specified in the third argument) determines whether the characters in the second argument are kept or removed from the result.</td>
</tr>
</tbody>
</table>

The COMPRESS function compiles a list of characters to keep or remove, comprising the characters in the second argument plus any types of characters that are specified by the modifiers. For example, the “d” or “D” modifier specifies digits. Both of the following function calls remove digits from the result:

```plaintext
COMPRESS(source, "1234567890");
COMPRESS(source, , "d");
```

To remove digits and plus or minus signs, you could use either of the following function calls:

```plaintext
COMPRESS(source, "1234567890+-");
COMPRESS(source, "+-", "d");
```

If the COMPRESS function returns a value to a variable that has not yet been assigned a length, then by default the variable length is determined by the length of the first argument.


Examples

Example 1: Compressing Blanks

SAS Statements | Results
---|---
```sas
a='AB C D '; b=compress(a); put b;
```

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a='AB C D '; b=compress(a); put b;</td>
<td>ABCD</td>
</tr>
</tbody>
</table>

Example 2: Compressing Lowercase Letters

SAS Statements | Results
---|---
```sas
x='123-4567-8901 B 234-5678-9012 c'; y=compress(x,'ABCD','l'); put y;
```

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x='123-4567-8901 B 234-5678-9012 c'; y=compress(x,'ABCD','l'); put y;</td>
<td>123-4567-8901 234-5678-9012</td>
</tr>
</tbody>
</table>

Example 3: Compressing Tab Characters

SAS Statements | Results
---|---
```sas
x='1 2 3 4 5'; y=compress(x,'s'); put y;
```

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x='1 2 3 4 5'; y=compress(x,'s'); put y;</td>
<td>12345</td>
</tr>
</tbody>
</table>

Example 4: Keeping Characters in the List

SAS Statements | Results
---|---
```sas
x='Math A English B Physics A'; y=compress(x,'ABCD','k'); put y;
```

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x='Math A English B Physics A'; y=compress(x,'ABCD','k'); put y;</td>
<td>ABA</td>
</tr>
</tbody>
</table>
See Also

Functions:
“COMPBL Function” on page 446
“LEFT Function” on page 625
“TRIM Function” on page 863

CONSTANT Function

Computes some machine and mathematical constants

Category: Mathematical

Syntax

CONSTANT(constant<, parameter>)

Arguments

constant
is a character string that identifies the constant. Valid constants are as follows:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>The natural base</td>
<td>‘E’</td>
</tr>
<tr>
<td>Euler constant</td>
<td>‘EULER’</td>
</tr>
<tr>
<td>Pi</td>
<td>‘PI’</td>
</tr>
<tr>
<td>Exact integer</td>
<td>‘EXACTINT’ &lt;, nbytes&gt;</td>
</tr>
<tr>
<td>The largest double-precision number</td>
<td>‘BIG’</td>
</tr>
<tr>
<td>The log with respect to base of BIG</td>
<td>‘LOGBIG’ &lt;,base&gt;</td>
</tr>
<tr>
<td>The square root of BIG</td>
<td>‘SQRTBIG’</td>
</tr>
<tr>
<td>The smallest double-precision number</td>
<td>‘SMALL’</td>
</tr>
<tr>
<td>The log with respect to base of SMALL</td>
<td>‘LOGSMALL’ &lt;,base&gt;</td>
</tr>
<tr>
<td>The square root of SMALL</td>
<td>‘SQRTSMALL’</td>
</tr>
<tr>
<td>Machine precision constant</td>
<td>‘MACEPS’</td>
</tr>
<tr>
<td>The log with respect to base of MACEPS</td>
<td>‘LOGMACEPS’ &lt;,base&gt;</td>
</tr>
<tr>
<td>The square root of MACEPS</td>
<td>‘SQRTMACEPS’</td>
</tr>
</tbody>
</table>

parameter
is an optional numeric parameter. Some of the constants specified in constant have an optional argument that alters the functionality of the CONSTANT function.
Details

The natural base

\textbf{CONSTANT('E')}

The natural base is described by the following equation:

\[
\lim_{x \to 0} (1 + x)^\frac{1}{x} \approx 2.718281828459045
\]

Euler constant

\textbf{CONSTANT('EULER')}

Euler’s constant is described by the following equation:

\[
\lim_{n \to \infty} \left\{ \sum_{j=1}^{n} \frac{1}{j} - \log(n) \right\} \approx 0.577215664901532860
\]

Pi

\textbf{CONSTANT('PI')}

Pi is the well-known constant in trigonometry that is the ratio between the circumference and the diameter of a circle. Many expressions exist for computing this constant. One such expression for the series is described by the following equation:

\[
4 \sum_{j=0}^{j=\infty} \frac{(-1)^j}{2j + 1} \approx 3.14159265358979323846
\]

Exact integer

\textbf{CONSTANT('EXACTINT' \(<,\ nbytes\)>)}

where

\textit{nbytes}

is a numeric value that is the number of bytes.

\textbf{Range:} \(2 \leq nbytes \leq 8\)

\textbf{Default:} 8

The exact integer is the largest integer \(k\) such that all integers less than or equal to \(k\) in absolute value have an exact representation in a SAS numeric variable of length \(nbytes\). This information can be useful to know before you trim a SAS numeric variable from the default 8 bytes of storage to a lower number of bytes to save storage.

The largest double-precision number

\textbf{CONSTANT('BIG')}

This case returns the largest double-precision floating-point number (8-bytes) that is representable on your computer.
The log with respect to base of BIG

CONSTANT('LOGBIG' <, base>)

where

base

is a numeric value that is the base of the logarithm.

Restriction: The base that you specify must be greater than the value of 1+SQRTMACEPS.

Default: the natural base, E.

This case returns the logarithm with respect to base of the largest double-precision floating-point number (8-bytes) that is representable on your computer.

It is safe to exponentiate the given base raised to a power less than or equal to CONSTANT('LOGBIG', base) by using the power operation (**). without causing any overflows.

It is safe to exponentiate any floating-point number less than or equal to CONSTANT('LOGBIG') by using the exponential function, EXP, without causing any overflows.

The square root of BIG

CONSTANT('SQRTBIG')

This case returns the square root of the largest double-precision floating-point number (8-bytes) that is representable on your computer.

It is safe to square any floating-point number less than or equal to CONSTANT('SQRTBIG') without causing any overflows.

The smallest double-precision number

CONSTANT('SMALL')

This case returns the smallest double-precision floating-point number (8-bytes) that is representable on your computer.

The log with respect to base of SMALL

CONSTANT('LOGSMALL' <, base>)

where

base

is a numeric value that is the base of the logarithm.

Restriction: The base that you specify must be greater than the value of 1+SQRTMACEPS.

Default: the natural base, E.

This case returns the logarithm with respect to base of the smallest double-precision floating-point number (8-bytes) that is representable on your computer.

It is safe to exponentiate the given base raised to a power greater than or equal to CONSTANT('LOGSMALL', base) by using the power operation (**). without causing any underflows or 0.

It is safe to exponentiate any floating-point number greater than or equal to CONSTANT('LOGSMALL') by using the exponential function, EXP, without causing any underflows or 0.

The square root of SMALL

CONSTANT('SQRTSMALL')
This case returns the square root of the smallest double-precision floating-point number (8-bytes) that is representable on the machine.

It is safe to square any floating-point number greater than or equal to \texttt{CONSTANT('SQRTBIG')} without causing any underflows or 0.

**Machine precision**

\texttt{CONSTANT('MACEPS')}

This case returns the smallest double-precision floating-point number (8-bytes) \( c = 2^{-j} \) for some integer \( j \), such that \( 1 + c > 1 \).

This constant is important in finite precision computations. A number \( n_1 \) is considered larger than another number \( n_2 \) if the (8-byte) representation of \( n_1 + n_2 \) is identical to \( n_r \). This constant can be used in summing series to implement a machine dependent stopping criterion.

**The log with respect to base of MACEPS**

\texttt{CONSTANT('LOGMACEPS' <, \textit{base}>)}

where

\begin{itemize}
  \item \textit{base} is a numeric value that is the base of the logarithm.
  \item \textbf{Restriction:} The \textit{base} that you specify must be greater than the value of \( 1+\text{SQRTMACEPS} \).
  \item \textbf{Default:} the natural base, \( E \).
\end{itemize}

This case returns the logarithm with respect to \textit{base} of \texttt{CONSTANT('MACEPS')}.

**The square root of MACEPS**

\texttt{CONSTANT('SQRTMACEPS')}

This case returns the square root of \texttt{CONSTANT('MACEPS')}.

---

### CONVX Function

**Returns the convexity for an enumerated cash flow**

**Category:** Financial

**Syntax**

\texttt{CONVX(y,f,c(1), ... ,c(k))}

**Arguments**

\begin{itemize}
  \item \( y \) specifies the effective per-period yield-to-maturity, expressed as a fraction.
    \item \textbf{Range:} \( 0 < y < 1 \)
  \item \( f \)
specifies the frequency of cash flows per period.

**Range:** \( f > 0 \)

c(1), ..., c(k)

specifies a list of cash flows.

**Details**

The CONVX function returns the value

\[
C = \frac{\sum_{k=1}^{K} k(k + f) \frac{c(k)}{(1 + fy)^{2}}}{P(1 + fy)^{2}}
\]

where

\[
P = \sum_{k=1}^{K} \frac{c(k)}{(1 + fy)^{2}}
\]

**Examples**

```plaintext
data _null_;  
c=convx(1/20,1,.33,.44,.55,.49,.50,.22,.4,.8,.01,.36,.2,.4);  
put c;  
run;
```

The value returned is 42.3778.

---

**CONVXP Function**

**Synopsis**

Returns the convexity for a periodic cash flow stream, such as a bond

**Category:** Financial

**Syntax**

`CONVXP(A,c,n,K,k0,y)`

**Arguments**

**A**

specifies the par value.

**Range:** \( A > 0 \)
\( c \)
specifies the nominal per-period coupon rate, expressed as a fraction.
**Range:** \( 0 \leq c < 1 \)

\( n \)
specifies the number of coupons per period.
**Range:** \( n > 0 \) and is an integer

\( K \)
specifies the number of remaining coupons.
**Range:** \( K > 0 \) and is an integer

\( k_0 \)
specifies the time from the present date to the first coupon date, expressed in terms of the number of periods.
**Range:** \( 0 < k_0 \leq \frac{1}{n} \)

\( y \)
specifies the nominal per-period yield-to-maturity, expressed as a fraction.
**Range:** \( y > 0 \)

**Details**
The CONVXP function returns the value

\[
C = \frac{1}{n^2} \left( \sum_{k=1}^{K} t_k \left( t_k + 1 \right) \frac{c(k)}{\left( 1 + \frac{y}{n} \right)^{t_k}} \right)
\]

where
\[
t_k = n k_0 + k - 1
\]
\[
c(k) = \frac{c}{n} A \quad \text{for} \quad k = 1, \ldots, K - 1
\]
\[
c(K) = \left( 1 + \frac{c}{n} \right) A
\]

and where

\[
P = \sum_{k=1}^{K} \frac{c(k)}{\left( 1 + \frac{y}{n} \right)^{t_k}}
\]

**Examples**
The following example demonstrates the use of CONVXP with a face value of 1000, an annual coupon rate of 0.01, 4 coupons per year, 14 remaining coupons, time from settlement date to next coupon is 0.165, and the price with accrued interest is 800.

```data _null_;
y=convxp(1000,.01,4,14,.33/2,800);
p=put y;
```
The value returned is 11.6023.

COS Function

Returns the cosine

Category: Trigonometric

Syntax

COS(argument)

Arguments

argument is numeric and is specified in radians.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=cos(0.5);</td>
<td>0.8775825619</td>
</tr>
<tr>
<td>x=cos(0);</td>
<td>1</td>
</tr>
<tr>
<td>x=cos(3.14159/3);</td>
<td>0.500000766</td>
</tr>
</tbody>
</table>

COSH Function

Returns the hyperbolic cosine

Category: Hyperbolic

Syntax

COSH(argument)
Arguments

**argument**
is numeric.

Details
The COSH function returns the hyperbolic cosine of the argument, given by

\[
\frac{e^{argument} + e^{-argument}}{2}
\]

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=cosh(0);</td>
<td>1</td>
</tr>
<tr>
<td>x=cosh(-5.0);</td>
<td>74.209948525</td>
</tr>
<tr>
<td>x=cosh(0.5);</td>
<td>1.1276259652</td>
</tr>
</tbody>
</table>

COUNT Function

Counts the number of times that a specific substring of characters appears within a character string that you specify

**Category:** Character

Syntax

COUNT(string,substring<,modifiers>)

Arguments

**string**
specifies a character constant, variable, or expression in which substrings are to be counted.

**Tip:** Enclose a literal string of characters in quotation marks.

**substring**
is a character constant, variable, or expression that specifies the substring of characters to count in **string**.

**Tip:** Enclose a literal string of characters in quotation marks.

**modifiers**
is a character constant, variable, or expression that specifies one or more modifiers. The following modifiers can be in uppercase or lowercase:

- **i** ignores character case during the count. If this modifier is not specified, COUNT only counts character substrings with the same case as the characters in *substring*.
- **t** trims trailing blanks from *string* and *substring*.

**Tip:** If the modifier is a constant, enclose it in quotation marks. Specify multiple constants in a single set of quotation marks. **Modifier** can also be expressed as a variable or an expression that evaluates to one or more constants.

**Details**

The COUNT function searches *string*, from left to right, for the number of occurrences of the specified *substring*, and returns that number of occurrences. If the substring is not found in *string*, COUNT returns a value of 0.

**CAUTION:**  
If two occurrences of the specified substring overlap in the string, inconsistent results will be returned. For example, COUNT('booboooboo', 'booboo') might return either a 1 or a 2.

**Comparisons**

The COUNT function counts substrings of characters in a character string, whereas the COUNTC function counts individual characters in a character string.

**Examples**

The following SAS statements produce these results:

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>xyz='This is a thistle? Yes, this is a thistle.‘; howmanythis=count(xyz,'this'); put howmanythis;</td>
<td>3</td>
</tr>
<tr>
<td>xyz='This is a thistle? Yes, this is a thistle.‘; howmanyis=count(xyz,'is'); put howmanyis;</td>
<td>6</td>
</tr>
<tr>
<td>howmanythis_i=count('This is a thistle? Yes, this is a thistle.‘,'this','i'); put howmanythis_i;</td>
<td>4</td>
</tr>
<tr>
<td>variable1='This is a thistle? Yes, this is a thistle.‘; variable2='is ‘; variable3='i'; howmanyis_i=count(variable1,variable2,variable3); put howmanyis_i;</td>
<td>4</td>
</tr>
<tr>
<td>expression1='This is a thistle? ‘</td>
<td></td>
</tr>
</tbody>
</table>
See Also

Functions:

“COUNTC Function” on page 468
“FIND Function” on page 532
“INDEX Function” on page 588
“RXMATCH Function” on page 801

COUNTC Function

Counts the number of specific characters that either appear or do not appear within a character string that you specify

Category: Character

Syntax

COUNTC(string, characters, modifiers)

Arguments

string
specifies a character constant, variable, or expression in which characters are to be counted.
Tip: Enclose a literal string of characters in quotation marks.

characters
is a character constant, variable, or expression that specifies one or more characters to count in string.
Tip: Enclose a literal string of characters in quotation marks.

modifiers
is a character constant, variable, or expression that specifies one or more modifiers. The following modifiers can be in uppercase or lowercase:

i ignores character case during the count. If this modifier is not specified, COUNTC only counts characters with the same case as the characters in characters.
o processes characters and modifiers only once, at the first call to this instance of COUNTC. Consequently, if you change the value of characters or modifiers in subsequent calls, the change is ignored by COUNTC.
t trims trailing blanks from string and characters.
v counts only the characters that do not appear in characters.

Tip: If the modifier is a constant, enclose it in quotation marks. Specify multiple constants in a single set of quotation marks. Modifier can also be expressed as a variable or an expression that evaluates to one or more constants.
Details
The COUNTC function searches *string* for all occurrences of the specified *characters* and returns the count for all of those characters. If none are found, COUNTC returns a value of 0.

Comparisons
The COUNTC function counts individual characters in a character string, whereas the COUNT function counts substrings of characters in a character string.

Examples
The following SAS statements produce these results:

```
SAS Statements Results
xyz='Baboons Eat Bananas '; howmanya=countc(xyz,'a'); put howmanya; 5
xyz='Baboons Eat Bananas '; howmanyb=countc(xyz,'b'); put howmanyb; 1
howmanyb_i=countc('Baboons Eat Bananas ','b','i'); put howmanyb_i; 3
xyz='Baboons Eat Bananas '; howmanyab_i=countc(xyz,'ab','i'); put howmanyab_i; 8
variable1='Baboons Eat Bananas '; variable2='ab'; variable3='iv'; howmanyab_iv=countc(variable1,variable2,variable3); put howmanyab_iv; 16
expression1='Baboons '||'Eat Bananas '; expression2=trim('ab '); expression3=compress('i '||'v '||' t'); howmanyab_ivt=countc(expression1,expression2,expression3); put howmanyab_ivt; 11
```

See Also
Functions:
“COUNT Function” on page 466
“FINDC Function” on page 534
“INDEXC Function” on page 589
“VERIFY Function” on page 883

CSS Function

Returns the corrected sum of squares
Category: Descriptive Statistics

Syntax

CSS(argument-1, argument-n)

Arguments

argument is numeric. At least one nonmissing argument is required. Otherwise, the function returns a missing value. If you have more than one argument, the argument list can consist of a variable list, which is preceded by OF.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1=css(5,9,3,6);</td>
<td>18.75</td>
</tr>
<tr>
<td>x2=css(5,8,9,6,.);</td>
<td>10</td>
</tr>
<tr>
<td>x3=css(8,9,6,.);</td>
<td>4.6666666667</td>
</tr>
<tr>
<td>x4=css(of x1-x3);</td>
<td>101.11574074</td>
</tr>
</tbody>
</table>

CUROBS Function

Returns the observation number of the current observation

Category: SAS File I/O

Syntax

CUROBS(data-set-id)

Arguments

data-set-id specifies the data set identifier that the OPEN function returns.

Details

CAUTION: Use this function only with an uncompressed SAS data set that is accessed using a native library engine.
If the engine being used does not support observation numbers, the function returns a missing value.

With a SAS data view, the function returns the relative observation number, that is, the number of the observation within the SAS data view (as opposed to the number of the observation within any related SAS data set).

**Examples**

This example uses the FETCHOBS function to fetch the tenth observation in the data set MYDATA. The value of OBSNUM returned by CUROBS is 10.

```sas
%let dsid=%sysfunc(open(mydata,i));
%let rc=%sysfunc(fetchobs(&dsid,10));
%let obsnum=%sysfunc(curobs(&dsid));
```

**See Also**

Functions:

- “FETCHOBS Function” on page 523
- “OPEN Function” on page 690

---

**CV Function**

Returns the coefficient of variation

**Category:** Descriptive Statistics

**Syntax**

```sas
CV(argument,argument, ...)
```

**Arguments**

`argument`

is numeric. At least two arguments are required. The argument list may consist of a variable list, which is preceded by OF.

**Examples**

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x1=cv(5,9,3,6);</code></td>
<td>43.47826087</td>
</tr>
<tr>
<td><code>x2=cv(5,8,9,6,.);</code></td>
<td>26.082026548</td>
</tr>
<tr>
<td><code>x3=cv(8,9,6,.);</code></td>
<td>19.924242152</td>
</tr>
<tr>
<td><code>x4=cv(of x1-x3);</code></td>
<td>40.953539216</td>
</tr>
</tbody>
</table>
**DACCDB Function**

**Returns the accumulated declining balance depreciation**

**Category:** Financial

**Syntax**

\[
\text{DACCDB}(p, v, y, r)
\]

**Arguments**

- **\( p \)**
  is numeric, the period for which the calculation is to be done. For noninteger \( p \) arguments, the depreciation is prorated between the two consecutive time periods that precede and follow the fractional period.

- **\( v \)**
  is numeric, the depreciable initial value of the asset.

- **\( y \)**
  is numeric, the lifetime of the asset.
  
  **Range:** \( y > 0 \)

- **\( r \)**
  is numeric, the rate of depreciation expressed as a decimal.
  
  **Range:** \( r > 0 \)

**Details**

The DACCDB function returns the accumulated depreciation by using a declining balance method. The formula is

\[
\text{DACCDB}(p, v, y, r) = \begin{cases} 
0 & \text{if } p \leq 0 \\
v \left(1 - \left(1 - \frac{r}{y}\right)^{\text{int}(p)}\right) \left(1 - \left(p - \text{int}(p)\right)\frac{r}{y}\right) & \text{if } p > 0
\end{cases}
\]

Note that \( \text{int}(p) \) is the integer part of \( p \). The \( p \) and \( y \) arguments must be expressed by using the same units of time. A double-declining balance is obtained by setting \( r \) equal to 2.

**Examples**

An asset has a depreciable initial value of $1000 and a fifteen-year lifetime. Using a 200 percent declining balance, the depreciation throughout the first 10 years can be expressed as

\[
a = \text{daccdb}(10, 1000, 15, 2);
\]

The value returned is 760.93. The first and the third arguments are expressed in years.
DACCDBSL Function

Returns the accumulated declining balance with conversion to a straight-line depreciation

Category: Financial

Syntax

DACCDBSL(p,v,y,r)

Arguments

\( p \)

is numeric, the period for which the calculation is to be done.

\( v \)

is numeric, the depreciable initial value of the asset.

\( y \)

is an integer, the lifetime of the asset.

Range: \( y > 0 \)

\( r \)

is numeric, the rate of depreciation that is expressed as a fraction.

Range: \( r > 0 \)

Details

The DACCDBSL function returns the accumulated depreciation by using a declining balance method, with conversion to a straight-line depreciation function that is defined by

\[
DACCDBSL(p,v,y,r) = \sum_{i=1}^{p} DEPDBSL(i,v,y,r)
\]

The declining balance with conversion to a straight-line depreciation chooses for each time period the method of depreciation (declining balance or straight-line on the remaining balance) that gives the larger depreciation. The \( p \) and \( y \) arguments must be expressed by using the same units of time.

Examples

An asset has a depreciable initial value of $1,000 and a ten-year lifetime. Using a declining balance rate of 150 percent, the accumulated depreciation of that asset in its fifth year can be expressed as

\[
y5=daccdbsl(5,1000,10,1.5);
\]

The value returned is 564.99. The first and the third arguments are expressed in years.
DACCSL Function

Returns the accumulated straight-line depreciation

Category: Financial

Syntax

DACCSL(p,v,y)

Arguments

\( p \)

is numeric, the period for which the calculation is to be done. For fractional \( p \), the depreciation is prorated between the two consecutive time periods that precede and follow the fractional period.

\( v \)

is numeric, the depreciable initial value of the asset.

\( y \)

is numeric, the lifetime of the asset.

Range: \( y > 0 \)

Details

The DACCSL function returns the accumulated depreciation by using the straight-line method, which is given by

\[
DACCSL(p, v, y) = \begin{cases} 
0 & \text{if } p < 0 \\
v \left(\frac{p}{y}\right) & \text{if } 0 \leq p \leq y \\
v & \text{if } p > y
\end{cases}
\]

The \( p \) and \( y \) arguments must be expressed by using the same units of time.

Example

An asset, acquired on 01APR86, has a depreciable initial value of $1000 and a ten-year lifetime. The accumulated depreciation in the value of the asset through 31DEC87 can be expressed as

\[ a = \text{daccsl}(1.75, 1000, 10); \]

The value returned is 175.00. The first and the third arguments are expressed in years.

DACCSYD Function

Returns the accumulated sum-of-years-digits depreciation
Category: Financial

Syntax
DACCSYD(p,v,y)

Arguments

\( p \)

is numeric, the period for which the calculation is to be done. For noninteger \( p \) arguments, the depreciation is prorated between the two consecutive time periods that precede and follow the fractional period.

\( v \)

is numeric, the depreciable initial value of the asset.

\( y \)

is numeric, the lifetime of the asset.

Range: \( y > 0 \)

Details
The DACCSYD function returns the accumulated depreciation by using the sum-of-years-digits method. The formula is

\[
DACCSYD(p,v,y) = \begin{cases} 
0 & \text{if } p < 0 \\
\frac{v \cdot \text{int}(p)(y - \text{int}(y))^2 + (p - \text{int}(p))(y - \text{int}(p))}{\text{int}(y)(y - \text{int}(y))^2 + (y - \text{int}(y))^2} & \text{if } 0 \leq p \leq y \\
\frac{v}{\text{int}(y)(y - \text{int}(y))^2 + (y - \text{int}(y))^2} & \text{if } p > y 
\end{cases}
\]

Note that \( \text{int}(y) \) indicates the integer part of \( y \). The \( p \) and \( y \) arguments must be expressed by using the same units of time.

Examples

An asset, acquired on 01OCT86, has a depreciable initial value of $1,000 and a five-year lifetime. The accumulated depreciation of the asset throughout 01JAN88 can be expressed as

\[
y2 = \text{daccsy}(15/12,1000,5);
\]

The value returned is 400.00. The first and the third arguments are expressed in years.

DACCTAB Function

Returns the accumulated depreciation from specified tables

Category: Financial
Syntax
DACCTAB(p,v,t1, ... ,tn)

Arguments

\( p \)
is numeric, the period for which the calculation is to be done. For noninteger \( p \) arguments, the depreciation is prorated between the two consecutive time periods that precede and follow the fractional period.

\( v \)
is numeric, the depreciable initial value of the asset.

\( t1,t2, \ldots ,tn \)
are numeric, the fractions of depreciation for each time period with \( t1+t2+...tn \leq 1 \).

Details
The DACCTAB function returns the accumulated depreciation by using user-specified tables. The formula for this function is

\[
DACCTAB(p,v,t_{1},t_{2},\ldots ,t_{n}) = \begin{cases} 
0 & p \leq 0 \\
v \left(t_{1} + t_{2} + \ldots + t_{\text{int}(p)} + (p - \text{int}(p)) t_{\text{int}(p) + 1}\right) & 0 < p < n \\
& p \geq n
\end{cases}
\]

For a given \( p \), only the arguments \( t_{1},t_{2},\ldots ,t_{k} \) need to be specified with \( k=\text{ceil}(p) \).

Examples
An asset has a depreciable initial value of $1000 and a five-year lifetime. Using a table of the annual depreciation rates of .15, .22, .21, .21, and .21 during the first, second, third, fourth, and fifth years, respectively, the accumulated depreciation throughout the third year can be expressed as

\[
y_{3}=\text{dacctab}(3,1000,.15,.22,.21,.21,.21);
\]

The value that is returned is 580.00. The fourth rate, .21, and the fifth rate, .21, can be omitted because they are not needed in the calculation.

DAIRY Function
Returns the derivative of the AIRY function

Category: Mathematical
**Syntax**

DAIRY(x)

**Arguments**

x

is numeric.

**Details**

The DAIRY function returns the value of the derivative of the AIRY function (Abramowitz and Stegun 1964; Amos, Daniel, and Weston 1977).

**Examples**

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=dairy(2.0);</td>
<td>-0.053090384</td>
</tr>
<tr>
<td>x=dairy(-2.0);</td>
<td>0.6182590207</td>
</tr>
</tbody>
</table>

**DATDIF Function**

Returns the number of days between two dates

**Category:** Date and Time

**Syntax**

DATDIF(sdate, edate, basis)

**Arguments**

sdate

specifies a SAS date value that identifies the starting date.

edate

specifies a SAS date value that identifies the ending date.

basis

identifies a character constant or variable that describes how SAS calculates the date difference. The following character strings are valid:

'30/360'

specifies a 30 day month and a 360 day year. Each month is considered to have 30 days, and each year 360 days, regardless of the actual number of days in each month or year.
DATE Function

Alias: ’360’

Tip: If either date falls at the end of a month, SAS treats the date as if it were the last day of a 30-day month.

‘ACT/ACT’
uses the actual number of days between dates.

Alias: ’Actual’

Examples

In the following example, DATDIF returns the actual number of days between two dates, and the number of days based on a 30-month and 360-day year.

```
data _null;
  sdate='16oct78'd;
  edate='16feb96'd;
  actual=datdif(sdate, edate, ’act/act’);
  days360=datdif(sdate, edate, ’30/360’);
  put actual= days360=;
run;
```

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>put actual=;</td>
<td>6332</td>
</tr>
<tr>
<td>put days360=;</td>
<td>6240</td>
</tr>
</tbody>
</table>

See Also

Functions:

“YRDIF Function” on page 918

DATE Function

Returns the current date as a SAS date value

Category: Date and Time

Alias: TODAY

See: “TODAY Function” on page 859

Syntax

DATE()
DATEJUL Function

Converts a Julian date to a SAS date value

Category:  Date and Time

Syntax

DATEJUL(julian-date)

Arguments

julian-date specifies a SAS numeric expression that represents a Julian date. A Julian date in SAS is a date in the form yyddd or yyyyddd, where yy or yyyy is a two-digit or four-digit integer that represents the year and ddd is the number of the day of the year. The value of ddd must be between 1 and 365 (or 366 for a leap year).

Examples

The following SAS statements produce these results:

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xstart=datejul(94365); put Xstart / Xstart date9.;</td>
<td>12783 31DEC1994</td>
</tr>
<tr>
<td>Xend=datejul(2001001); put Xend / Xend date9.;</td>
<td>14976 01JAN2001</td>
</tr>
</tbody>
</table>

See Also

Function:
“JULDATE Function” on page 613

DATEPART Function

Extracts the date from a SAS datetime value

Category:  Date and Time
**Syntax**

`DATEPART(datetime)`

**Arguments**

`datetime` specifies a SAS expression that represents a SAS datetime value.

**Examples**

The following SAS statements produce this result:

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn='01feb94:8:45'dt; servdate=datepart(conn); put servdate worddate.;</td>
<td>February 1, 1994</td>
</tr>
</tbody>
</table>

**See Also**

Functions:
- “DATETIME Function” on page 480
- “TIMEPART Function” on page 856
See Also

Functions:
“DATE Function” on page 478
“TIME Function” on page 856

DAY Function

Returns the day of the month from a SAS date value
Category: Date and Time

Syntax
DAY(date)

Arguments

date
specifies a SAS expression that represents a SAS date value.

Details
The DAY function produces an integer from 1 to 31 that represents the day of the month.

Examples
The following SAS statements produce this result:

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>now='05may97'd;</td>
<td></td>
</tr>
<tr>
<td>d=day(now);</td>
<td></td>
</tr>
<tr>
<td>put d;</td>
<td>5</td>
</tr>
</tbody>
</table>

See Also

Functions:
“MONTH Function” on page 655
“YEAR Function” on page 916

DCLOSE Function

Closes a directory that was opened by the DOPEN function
Category: External Files

Syntax

**DCLOSE**(directory-id)

Argument

directory-id
specifies the identifier that was assigned when the directory was opened, generally by the DOPEN function.

Details

DCLOSE returns 0 if the operation was successful, ≠0 if it was not successful. The DCLOSE function closes a directory that was previously opened by the DOPEN function. DCLOSE also closes any open members.

*Note:* All directories or members opened within a DATA step are closed automatically when the DATA step ends.

Examples

Example 1: Using DCLOSE to Close a Directory
This example opens the directory to which the fileref MYDIR has previously been assigned, returns the number of members, and then closes the directory:

```%macro memnum(filrf,path);
%let rc=%sysfunc(filename(filrf,&path));
%if %sysfunc(fileref(&filrf)) = 0 %then
%do;
/* Open the directory. */
%let did=%sysfunc(dopen(&filrf));
%put did=&did;
/* Get the member count. */
%let memcount=%sysfunc(dnum(&did));
%put &memcount members in &filrf.;
/* Close the directory. */
%let rc= %sysfunc(dclose(&did));
%end;
%else %put Invalid FILEREF;
%mend;
%memnum(MYDIR,physical-filename)
```

Example 2: Using DCLOSE within a DATA Step
This example uses the DCLOSE function within a DATA step:

```%let filrf=MYDIR;
data _null_
  rc=filename("&filrf","physical-filename");
  if fileref("&filrf") = 0 then
do;
  /* Open the directory. */```
did=dopen("&filrf");
    /* Get the member count. */
    memcount=dnum(did);
    put memcount "members in &filrf";
    /* Close the directory. */
    rc=dclose(did);
end;
else put "Invalid FILEREF";
run;

See Also

Functions:
   “DOPEN Function” on page 502
   “FCLOSE Function” on page 518
   “FOPEN Function” on page 549
   “MOPEN Function” on page 656

DCREATE Function

Creates an external directory

Category: External Files

Syntax

new-directory=DCREATE(directory-name<,parent-directory>)

Arguments

new-directory
   contains the complete pathname of the new directory, or contains an empty string if
   the directory cannot be created.

directory-name
   specifies the name of the directory to create. This value cannot include a pathname.

parent-directory
   contains the complete pathname of the directory in which to create the new directory.
   If you do not supply a value for parent-directory, then the current directory is the
   parent directory.

Details

The DCREATE function enables you to create a directory in your operating
environment.
Examples

To create a new directory in the UNIX operating environment, using the name that is stored in the variable DirectoryName, follow this form:

\[ \text{NewDirectory}=\text{dcreate} \left( \text{DirectoryName}, '/local/u/abcdef/\right) \];

To create a new directory in the Windows operating environment, using the name that is stored in the variable DirectoryName, follow this form:

\[ \text{NewDirectory}=\text{dcreate} \left( \text{DirectoryName}, 'd:\testdir/\right) \];

**DEPDB Function**

Returns the declining balance depreciation

**Category:** Financial

**Syntax**

\[ \text{DEPDB}(p,v,y,r) \]

**Arguments**

\( p \)

is numeric, the period for which the calculation is to be done. For noninteger \( p \) arguments, the depreciation is prorated between the two consecutive time periods that precede and follow the fractional period.

\( v \)

is numeric, the depreciable initial value of the asset.

\( y \)

is numeric, the lifetime of the asset.

**Range:** \( y > 0 \)

\( r \)

is numeric, the rate of depreciation that is expressed as a fraction.

**Range:** \( r \geq 0 \)

**Details**

The DEPDB function returns the depreciation by using the declining balance method, which is given by

\[
\text{DEPDB}(p,v,y,r) = \text{DACCDB}(p,v,y,r) - \text{DACCDB}(p-1,v,y,r)
\]

The \( p \) and \( y \) arguments must be expressed by using the same units of time. A double-declining balance is obtained by setting \( r \) equal to 2.
Examples

An asset has an initial value of $1,000 and a fifteen-year lifetime. Using a declining balance rate of 200 percent, the depreciation of the value of the asset for the tenth year can be expressed as

\[ y_{10} = \text{DEPDBSL}(10, 1000, 15, 2); \]

The value returned is 36.78. The first and the third arguments are expressed in years.

DEPDBSL Function

Returns the declining balance with conversion to a straight-line depreciation

Category: Financial

Syntax

DEPDBSL(p, v, y, r)

Arguments

\( p \)

is an integer, the period for which the calculation is to be done.

\( v \)

is numeric, the depreciable initial value of the asset.

\( y \)

is an integer, the lifetime of the asset.

Range: \( y > 0 \)

\( r \)

is numeric, the rate of depreciation that is expressed as a fraction.

Range: \( r \geq 0 \)

Details

The DEPDBSL function returns the depreciation by using the declining balance method with conversion to a straight-line depreciation, which is given by

\[
\text{DEPDBSL}(p, v, y, r) = \begin{cases} 
0 & p < 0 \\
v \frac{r}{y} \left(1 - \frac{r}{y}\right)^{p-1} & p \leq t \\
v \left(1 - \frac{r}{y}\right)^{t} \frac{1 - \frac{p}{y}}{(y-t)} & p > t \\
0 & p > y 
\end{cases}
\]
where

\[ t = \text{int} \left( y - \frac{y}{r} + 1 \right) \]

and \( \text{int}(\cdot) \) denotes the integer part of a numeric argument.

The \( p \) and \( y \) arguments must be expressed by using the same units of time. The declining balance that changes to a straight-line depreciation chooses for each time period the method of depreciation (declining balance or straight-line on the remaining balance) that gives the larger depreciation.

**Examples**

An asset has a depreciable initial value of $1,000 and a ten-year lifetime. Using a declining balance rate of 150 percent, the depreciation of the value of the asset in the fifth year can be expressed as

\[ y_5 = \text{depdbsl}(5, 1000, 10, 1.5); \]

The value returned is 87.00. The first and the third arguments are expressed in years.

---

**DEPSL Function**

**Returns the straight-line depreciation**

**Category:** Financial

**Syntax**

DEPSL\((p,v,y)\)

**Arguments**

\( p \)

is numeric, the period for which the calculation is to be done. For fractional \( p \), the depreciation is prorated between the two consecutive time periods that precede and follow the fractional period.

\( v \)

is numeric, the depreciable initial value of the asset.

\( y \)

is numeric, the lifetime of the asset.

**Range:** \( y > 0 \)

**Details**

The DEPSL function returns the straight-line depreciation, which is given by
DEPSYD Function

Returns the sum-of-years-digits depreciation

Category: Financial

Syntax

DEPSYD(p,v,y)

Arguments

\( p \)

is numeric, the period for which the calculation is to be done. For noninteger \( p \) arguments, the depreciation is prorated between the two consecutive time periods that precede and follow the fractional period.

\( v \)

is numeric, the depreciable initial value of the asset.

\( y \)

is numeric, the lifetime of the asset in number of depreciation periods.

Range: \( y > 0 \)

Details

The DEPSYD function returns the sum-of-years-digits depreciation, which is given by

\[
\text{DEPSYD}(p,v,y) = \text{DACCSYD}(p,v,y) - \text{DACCSYD}(p-1,v,y)
\]

The \( p \) and \( y \) arguments must be expressed by using the same units of time.
Examples

An asset, acquired on 01OCT86, has a depreciable initial value of $1,000 and a five-year lifetime. The depreciations in the value of the asset for the years 1986 and 1987 can be expressed as

\[ y_1 = \text{depsyd}(3/12, 1000, 5); \]
\[ y_2 = \text{depsyd}(15/12, 1000, 5); \]

The values returned are 83.33 and 316.67, respectively. The first and the third arguments are expressed in years.

DEPTAB Function

Returns the depreciation from specified tables

Category: Financial

Syntax

\[ \text{DEPTAB}(p,v,t_1,...,t_n) \]

Arguments

\( p \)

is numeric, the period for which the calculation is to be done. For noninteger \( p \) arguments, the depreciation is prorated between the two consecutive time periods that precede and follow the fractional period.

\( v \)

is numeric, the depreciable initial value of the asset.

\( t_1, t_2, \ldots, t_n \)

are numeric, the fractions of depreciation for each time period with \( t_1 + t_2 + \ldots + t_n \leq 1 \).

Details

The DEPTAB function returns the depreciation by using specified tables. The formula is

\[
\text{DEPTAB}(p,v,t_1,t_2,...,t_n) = \text{DACCTAB}(p,v,t_1,t_2,...,t_n) - \text{DACCTAB}(p-1,v,t_1,t_2,...,t_n)
\]

For a given \( p \), only the arguments \( t_1,t_2,\ldots,t_k \) need to be specified with \( k=\text{ceil}(p) \).

Examples

An asset has a depreciable initial value of $1,000 and a five-year lifetime. Using a table of the annual depreciation rates of .15, .22, .21, .21, and .21 during the first,
second, third, fourth, and fifth years, respectively, the depreciation in the third year can
be expressed as

\[ y_3 = deplab(3, 1000, .15, .22, .21, .21, .21); \]

The value that is returned is 210.00. The fourth rate, .21, and the fifth rate, .21, can be
omitted because they are not needed in the calculation.

**DEQUOTE Function**

Removes matching quotation marks from a character string that begins with an individual
quotation mark and deletes everything that is to the right of the closing quotation mark

**Category:** Character

**Syntax**

DEQUOTE(string)

**Arguments**

string

specifies a character constant, variable, or expression.

**Details**

If the DEQUOTE function returns a value to a variable that has not yet been assigned
a length, then by default the variable length is determined by the length of the first
argument.

The value that is returned by the DEQUOTE function depends on the first character
or the first two characters in string:

- If the first character of string is not a single or double quotation mark, DEQUOTE
  returns string unchanged.
- If the first two characters of string are both single quotation marks or both double
  quotation marks, DEQUOTE returns a result with a length of zero.
- If the first character of string is a single quotation mark, the DEQUOTE function
  removes that single quotation mark from the result. DEQUOTE then scans string
  from left to right, looking for more single quotation marks. All paired single
  quotation marks are reduced to one single quotation mark. The first non-paired
  single quotation mark in string is removed and all characters to the right of that
  quotation mark are also removed.
- If the first character of string is a double quotation mark, the DEQUOTE function
  removes that double quotation mark from the result. DEQUOTE then scans string
  from left to right, looking for more double quotation marks. All paired double
  quotation marks are reduced to one double quotation mark. The first non-paired
  double quotation mark in string is removed and all characters to the right of that
  quotation mark are also removed.
Note: If string is a constant enclosed by quotation marks, those quotation marks are not part of the value of string. Therefore, you do not need to use DEQUOTE to remove the quotation marks that denote a constant.

Examples

This example demonstrates the use of DEQUOTE within a DATA step.

data test;
  input string $60.;
  result = dequote(string);
  datalines;
No quotes, no change
No "leading" quote, no change
"" returns a string with length zero
"Matching double quotes are removed"
‘Matching single quotes are removed’
"Paired "quotes"" are reduced"
‘Paired ‘quotes’’ are reduced’
"Single ‘quotes’ inside ‘‘ double’’ quotes are unchanged"
‘Double "quotes" inside "‘single’’ quotes are unchanged’
"No matching quote, no problem
Don’t remove this apostrophe
"Text after the matching quote" is "deleted"
;

proc print noobs;
title 'Input Strings and Output Results from DEQUOTE';
run;
Functions and CALL Routines

DEVIANCE Function

Computes the deviance
Category: Mathematical

Syntax

\[
\text{DEVIANCE}(\text{distribution, variable, shape-parameter(s)}<,>)
\]

Arguments

distribution
is a character string that identifies the distribution. Valid distributions are

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bernoulli</td>
<td>‘BERNOULLI’</td>
</tr>
<tr>
<td>Binomial</td>
<td>‘BINOMIAL’</td>
</tr>
<tr>
<td>Gamma</td>
<td>‘GAMMA’</td>
</tr>
<tr>
<td>Inverse Gauss (Wald)</td>
<td>‘IGAUSS’</td>
</tr>
</tbody>
</table>
variable
  is a numeric random variable.

shape-parameter(s)
  are one or more distribution-specific numeric parameters that characterize the shape
  of the distribution.

$\varepsilon$
  is an optional numeric small value used for all of the distributions, except for the
  normal distribution.

Details

The Bernoulli Distribution

\[
\text{DEVIANCE}('BERNOULLI', \text{variable}, p, \varepsilon)
\]

where

\begin{itemize}
  \item \text{variable}
    \begin{itemize}
      \item is a binary numeric random variable that has the value of 1 for success and 0 for
            failure.
    \end{itemize}
  \item $p$
    \begin{itemize}
      \item is a numeric probability of success with $\varepsilon \leq p \leq 1 - \varepsilon$.
    \end{itemize}
  \item $\varepsilon$
    \begin{itemize}
      \item is an optional positive numeric value that is used to bound $p$. Any value of $p$ in
            the interval $0 \leq p \leq \varepsilon$ is replaced by $\varepsilon$. Any value of $p$ in
            the interval $1 - \varepsilon \leq p \leq 1$ is replaced by $1 - \varepsilon$.
    \end{itemize}
\end{itemize}

The DEVIANCE function returns the deviance from a Bernoulli distribution with a
probability of success $p$, where success is defined as a random variable value of 1. The
equation follows:

\[
\text{DEVIANCE}('BERN', \text{variable}, p, \varepsilon) = \begin{cases} 
  -2 \log (1 - p) & x = 0 \\
  -2 \log (p) & x = 1 \\
  \text{otherwise}
\end{cases}
\]

The Binomial Distribution

\[
\text{DEVIANCE}('BINO', \text{variable}, \mu, n, \varepsilon)
\]

where

\begin{itemize}
  \item \text{variable}
    \begin{itemize}
      \item is a numeric random variable that contains the number of successes.
    \end{itemize}
  \item $\mu$
    \begin{itemize}
      \item is a numeric mean parameter.
    \end{itemize}
  \end{itemize}

\textbf{Range: } 0 \leq \text{variable} \leq 1
Functions and CALL Routines  \( \triangle \)  DEVIANCE Function 493

Range:  \( n \varepsilon \leq \mu \leq n(1-\varepsilon) \)

\( n \)

is an integer number of Bernoulli trials parameter

Range:  \( n \geq 0 \)

\( \varepsilon \)

is an optional positive numeric value that is used to bound \( \mu \). Any value of \( \mu \) in the interval \( 0 \leq \mu \leq n \varepsilon \) is replaced by \( n \varepsilon \). Any value of \( \mu \) in the interval \( n(1-\varepsilon) \leq \mu \leq n \) is replaced by \( n(1-\varepsilon) \).

The DEVIANCE function returns the deviance from a binomial distribution, with a probability of success \( p \), and a number of independent Bernoulli trials \( n \). The following equation describes the DEVIANCE function for the Binomial distribution, where \( x \) is the random variable.

\[
\text{DEVIANCE('BINO', } x, \mu, n) = \begin{cases} 
2 \left( \log \left( \frac{x}{\mu} \right) + (n - x) \log \left( \frac{n-x}{n-\mu} \right) \right) & x < 0 \\
0 & 0 \leq x \leq n \\
x > n 
\end{cases}
\]

The Gamma Distribution

\[
\text{DEVIANCE('GAMMA', } \text{variable, } \mu <, \varepsilon>)
\]

where

\( \text{variable} \)

is a numeric random variable.

Range:  \( \text{variable} \geq \varepsilon \)

\( \mu \)

is a numeric mean parameter.

Range:  \( \mu \geq \varepsilon \)

\( \varepsilon \)

is an optional positive numeric value that is used to bound \( \text{variable} \) and \( \mu \). Any value of \( \text{variable} \) in the interval \( 0 \leq \text{variable} \leq \varepsilon \) is replaced by \( \varepsilon \). Any value of \( \mu \) in the interval \( 0 \leq \mu \leq \varepsilon \) is replaced by \( \varepsilon \).

The DEVIANCE function returns the deviance from a gamma distribution with a mean parameter \( \mu \). The following equation describes the DEVIANCE function for the gamma distribution, where \( x \) is the random variable:

\[
\text{DEVIANCE('GAMMA', } x, \mu) = \begin{cases} 
2 \left( -\log \left( \frac{x}{\mu} \right) + \frac{x-\mu}{\mu} \right) & x < 0 \\
x \geq \varepsilon, \mu \geq \varepsilon
\end{cases}
\]

The Inverse Gauss (Wald) Distribution

\[
\text{DEVIANCE('IGAUSS' | 'WALD', } \text{variable, } \mu <, \varepsilon>)
\]

where

\( \text{variable} \)

is a numeric random variable.

Range:  \( \text{variable} \geq \varepsilon \)
is a numeric mean parameter.

Range: \( \mu \geq \varepsilon \)

\( \varepsilon \) is an optional positive numeric value that is used to bound \( \text{variable} \) and \( \mu \). Any value of \( \text{variable} \) in the interval \( 0 \leq \text{variable} \leq \varepsilon \) is replaced by \( \varepsilon \). Any value of \( \mu \) in the interval \( 0 \leq \mu \leq \varepsilon \) is replaced by \( \varepsilon \).

The DEVIANCE function returns the deviance from an inverse Gaussian distribution with a mean parameter \( \mu \). The following equation describes the DEVIANCE function for the inverse Gaussian distribution, where \( x \) is the random variable:

\[
\text{DEVIANCE}(\text{IGAUSS'}, x, \mu) = \begin{cases} 
\frac{(x-\mu)^2}{\mu^2 x} & x < 0 \\
0 & x \geq \varepsilon, \mu \geq \varepsilon
\end{cases}
\]

The Normal Distribution

\[
\text{DEVIANCE}('NORMAL' | 'GAUSSIAN', \text{variable}, \mu)
\]

where

\( \text{variable} \)

is a numeric random variable.

\( \mu \)

is a numeric mean parameter.

The DEVIANCE function returns the deviance from a normal distribution with a mean parameter \( \mu \). The following equation describes the DEVIANCE function for the normal distribution, where \( x \) is the random variable:

\[
\text{DEVIANCE}(\text{NORMAL'}, x, \mu) = (x - \mu)^2
\]

The Poisson Distribution

\[
\text{DEVIANCE}('POISSON', \text{variable}, \mu, <, \varepsilon>)
\]

where

\( \text{variable} \)

is a numeric random variable.

Range: \( \text{variable} \geq 0 \)

\( \mu \)

is a numeric mean parameter.

Range: \( \mu \geq \varepsilon \)

\( \varepsilon \)

is an optional positive numeric value that is used to bound \( \mu \). Any value of \( \mu \) in the interval \( 0 \leq \mu \leq \varepsilon \) is replaced by \( \varepsilon \).

The DEVIANCE function returns the deviance from a Poisson distribution with a mean parameter \( \mu \). The following equation describes the DEVIANCE function for the Poisson distribution, where \( x \) is the random variable:
DEVIANCE \left( ^t \text{POISSON}^t, x, \mu \right) = \begin{cases} 2 \left( x \log \left( \frac{x}{\mu} \right) - (x - \mu) \right) & x < 0 \\ x \geq 0, \mu \geq \epsilon \end{cases}

### DHMS Function

Returns a SAS datetime value from date, hour, minute, and second

**Category:** Date and Time

**Syntax**

\[ \text{DHMS}(\text{date}, \text{hour}, \text{minute}, \text{second}) \]

**Arguments**

- **date**
  - specifies a SAS expression that represents a SAS date value.

- **hour**
  - is numeric.

- **minute**
  - is numeric.

- **second**
  - is numeric.

**Details**

The DHMS function returns a numeric value that represents a SAS datetime value. This numeric value can be either positive or negative.

**Examples**

The following SAS statements produce these results:

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>dtid=dhms(’01jan03’d,15,30,15); put dtid; put dtid datetime.;</td>
<td>1357054215 01JAN03:15:30:15</td>
</tr>
<tr>
<td>dtid2=dhms(’01jan03’d,15,30,61); put dtid2; put dtid2 datetime.;</td>
<td>1357054261 01JAN03:15:31:01</td>
</tr>
<tr>
<td>dtid3=dhms(’01jan03’d,15,.5,15); put dtid3; put dtid3 datetime.;</td>
<td>1357052445 01JAN02:15:00:45</td>
</tr>
</tbody>
</table>
The following SAS statements show how to combine a SAS date value with a SAS time value into a SAS datetime value. If you execute these statements on April 2, 2003 at the time of 15:05:02, it produces these results:

```
SAS Statements Result
day=date();
time=time();
sasdt=dhms(day,0,0,time);
put sasdt datetime.; 02APR03:15:05:02
```

See Also

Function:
“HMS Function” on page 578

DIF Function

Returns differences between the argument and its $n$th lag

Category: Special

Syntax

\[ \text{DIF}(<n>(\text{argument})) \]

Arguments

$n$

specifies the number of lags.

argument

is numeric.

Details

The DIF functions, DIF1, DIF2, ..., DIF100, return the first differences between the argument and its $n$th lag. DIF1 can also be written as DIF. DIF$n$ is defined as

\[ DIF^n(x) = x - LAGn(x). \]

For details on storing and returning values from the LAG$n$ queue, see the LAG function.

Comparisons

The function DIF2(X) is not equivalent to the second difference DIF(DIF(X)).
Examples

This example demonstrates the difference between the LAG and DIF functions.

```sas
data two;
  input X @@;
  Z=lag(x);
  D=dif(x);
datalines;
1 2 6 4 7
;
proc print data=two;
run;
```

Results of the PROC PRINT step follow:

<table>
<thead>
<tr>
<th>OBS</th>
<th>X</th>
<th>Z</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>6</td>
<td>-2</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

See Also

Function:

“LAG Function” on page 620

---

**DIGAMMA Function**

Returns the value of the Digamma function

**Category:** Mathematical

**Syntax**

`DIGAMMA(argument)`

**Arguments**

`argument` is numeric.

**Restriction:** Nonpositive integers are invalid.
Details
The DIGAMMA function returns the ratio that is given by

\[ \Psi(x) = \frac{\Gamma'(x)}{\Gamma(x)} \]

where \( \Gamma(.) \) and \( \Gamma'(.) \) denote the Gamma function and its derivative, respectively. For \( \text{argument}>0 \), the DIGAMMA function is the derivative of the LGAMMA function.

Example

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=digamma(1.0);</td>
<td>-0.577215665</td>
</tr>
</tbody>
</table>

DIM Function

Returns the number of elements in an array

Category: Array

Syntax

\[
\text{DIM}\langle n\rangle\text{array-name)  \\
\text{DIM(array-name,bound-n)}
\]

Arguments

\( n \)
specifies the dimension, in a multidimensional array, for which you want to know the number of elements. If no \( n \) value is specified, the DIM function returns the number of elements in the first dimension of the array.

\( \text{array-name} \)
specifies the name of an array that was previously defined in the same DATA step.

\( \text{bound-n} \)
specifies the dimension, in a multidimensional array, for which you want to know the number of elements. Use \( \text{bound-n} \) only when \( n \) is not specified.

Details
The DIM function returns the number of elements in a one-dimensional array or the number of elements in a specified dimension of a multidimensional array when the lower bound of the dimension is 1. Use DIM in array processing to avoid changing the upper bound of an iterative DO group each time you change the number of array elements.
Comparisons

- DIM always returns a total count of the number of elements in an array dimension.
- HBOUND returns the literal value of the upper bound of an array dimension.

**Note:** This distinction is important when the lower bound of an array dimension has a value other than 1 and the upper bound has a value other than the total number of elements in the array dimension.

Examples

**Example 1: One-dimensional Array**  
In this example, DIM returns a value of 5. Therefore, SAS repeats the statements in the DO loop five times.

```sas
array big{5} weight sex height state city;
do i=1 to dim(big);
    more SAS statements;
end;
```

**Example 2: Multidimensional Array**  
This example shows two ways of specifying the DIM function for multidimensional arrays. Both methods return the same value for DIM, as shown in the table that follows the SAS code example.

```sas
array mult{5,10,2} mult1-mult100;
```

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Alternative Syntax</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIM(MULT)</td>
<td>DIM(MULT,1)</td>
<td>5</td>
</tr>
<tr>
<td>DIM2(MULT)</td>
<td>DIM(MULT,2)</td>
<td>10</td>
</tr>
<tr>
<td>DIM3(MULT)</td>
<td>DIM(MULT,3)</td>
<td>2</td>
</tr>
</tbody>
</table>

See Also

Functions:
- “HBOUND Function” on page 576
- “LBOUND Function” on page 624

Statements:
- “ARRAY Statement” on page 1102
- “Array Reference Statement” on page 1107
- “Array Processing” in *SAS Language Reference: Concepts*

**DINFO Function**

Returns information about a directory
Category: External Files

See: DINFO Function in the documentation for your operating environment.

Syntax

DINFO(directory-id,info-item)

Arguments

directory-id
specifies the identifier that was assigned when the directory was opened, generally by the DOPEN function.

info-item
specifies the information item to be retrieved. DINFO returns a blank if the value of the info-item argument is invalid. The information available varies according to the operating environment. This is a character value.

Details

Use DOPTNAME to determine the names of the available system-dependent directory information items. Use DOPTNUM to determine the number of directory information items available.

Operating Environment Information: DINFO returns the value of a system-dependent directory parameter. See the SAS documentation for your operating environment for information about system-dependent directory parameters.

Examples

Example 1: Using DINFO to Return Information about a Directory
This example opens the directory MYDIR, determines the number of directory information items available, and retrieves the value of the last one:

```sas
%let filrf=MYDIR;
%let rc=%sysfunc(filename(filrf,physical-name));
%let did=%sysfunc(dopen(&filrf));
%let numopts=%sysfunc(doptnum(&did));
%let foption=%sysfunc(doptname(&did,&numopts));
%let charval=%sysfunc(dinfo(&did,&foption));
%let rc=%sysfunc(dclose(&did));
```

Example 2: Using DINFO within a DATA Step
This example creates a data set that contains the name and value of each directory information item:

```sas
data diropts;
  length foption $ 12 charval $ 40;
  keep foption charval;
  rc=filename("mydir","physical-name");
  did=dopen("mydir");
  numopts=doptnum(did);
  do i=1 to numopts;
    foption=doptname(did,i);
charval=dinfo(did,foption);
output;
edn;
run;

See Also

Functions:
“DOPEN Function” on page 502
“DOPTNAME Function” on page 503
“DOPTNUM Function” on page 505
“FINO Function” on page 538
“FOPTNAME Function” on page 551
“FOPTNUM Function” on page 553

DNUM Function

Returns the number of members in a directory

Category: External Files

Syntax

DNUM(directory-id)

Argument

directory-id
specifies the identifier that was assigned when the directory was opened, generally by the DOPEN function.

Details

You can use DNUM to determine the highest possible member number that can be passed to DREAD.

Examples

Example 1: Using DNUM to Return the Number of Members This example opens the directory MYDIR, determines the number of members, and closes the directory:

%let filrf=MYDIR;
%let rc=%sysfunc(filename(filrf,physical-name));
%let did=%sysfunc(dopen(&filrf));
%let memcount=%sysfunc(dnum(&did));
%let rc=%sysfunc(dclose(&did));
Example 2: Using DNUM within a DATA Step  
This example creates a DATA step that returns the number of members in a directory called MYDIR:

```sas
data _null_;  
  rc=filename("mydir","physical-name");  
  did=dopen("mydir");  
  memcount=dnum(did);  
  rc=dclose(did);  
run;
```

See Also

Functions:
- “DOPEN Function” on page 502
- “DREAD Function” on page 506

### DOPEN Function

Opens a directory and returns a directory identifier value

**Category:** External Files

**See:** DOPEN Function in the documentation for your operating environment.

**Syntax**

DOPEN(**fileref**)

**Argument**

**fileref**

specifies the fileref assigned to the directory.

**Restriction:** You must associate a fileref with the directory before calling DOPEN.

**Details**

DOPEN opens a directory and returns a directory identifier value (a number greater than 0) that is used to identify the open directory in other SAS external file access functions. If the directory could not be opened, DOPEN returns 0. The directory to be opened must be identified by a fileref. You can assign filerefs using the FILENAME statement or the FILENAME external file access function. Under some operating environments, you can also assign filerefs using system commands.

**Operating Environment Information:** The term directory used in the description of this function and related SAS external file access functions refers to an aggregate grouping of files managed by the operating environment. Different operating environments identify such groupings with different names, such as directory, subdirectory, MACLIB, or partitioned data set. For details, see the SAS documentation for your operating environment.
Examples

Example 1: Using DOPEN to Open a Directory  This example assigns the fileref MYDIR to a directory. It uses DOPEN to open the directory. DOPTNUM determines the number of system-dependent directory information items available, and DCLOSE closes the directory:

   %let filrf=MYDIR;
   %let rc=%sysfunc(filename(filrf,physical-name));
   %let did=%sysfunc(dopen(&filrf));
   %let infocnt=%sysfunc(doptnum(&did));
   %let rc=%sysfunc(dclose(&did));

Example 2: Using DOPEN within a DATA Step  This example opens a directory for processing within a DATA step.

    data _null_;
      drop rc did;
      rc=filename("mydir","physical-name");
      did=dopen("mydir");
      if did > 0 then do;
        ...more statements...
      end;
    run;

See Also

Functions:
   “DCLOSE Function” on page 481
   “DOPTNUM Function” on page 505
   “FOPEN Function” on page 549
   “MOPEN Function” on page 656

DOPTNAME Function

Returns directory attribute information

Category:  External Files
See:  DOPTNAME Function in the documentation for your operating environment.

Syntax

DOPTNAME(directory-id,nval)
Arguments

directory-id
  specifies the identifier that was assigned when the directory was opened, generally by the DOPEN function.

Restriction: The directory must have been previously opened by using DOPEN.

nval
  specifies the sequence number of the option.

Details

Operating Environment Information: The number, names, and nature of the directory information varies between operating environments. The number of options that are available for a directory varies depending on the operating environment. For details, see the SAS documentation for your operating environment.

Examples

Example 1: Using DOPTNAME to Retrieve Directory Attribute Information  This example opens the directory with the fileref MYDIR, retrieves all system-dependent directory information items, writes them to the SAS log, and closes the directory:

```sas
%let filrf=mydir;
%let rc=%sysfunc(filename(filrf,physical-name));
%let did=%sysfunc(dopen(&filrf));
%let infocnt=%sysfunc(doptnum(&did));
%do j=1 %to &infocnt;
  %let opt=%sysfunc(doptname(&did,&j));
  %put Directory information=&opt;
%end;
%let rc=%sysfunc(dclose(&did));
```

Example 2: Using DOPTNAME within a DATA Step  This example creates a data set that contains the name and value of each directory information item:

```sas
data diropts;
  length optname $ 12 optval $ 40;
  keep optname optval;
  rc=filename("mydir","physical-name");
  did=dopen("mydir");
  numopts=doptnum(did);
  do i=1 to numopts;
    optname=doptname(did,i);
    optval=dinfo(did,optname);
    output;
  end;
run;
```

See Also

Functions:

“DINFO Function” on page 499
DOPTNUM Function

Returns the number of information items that are available for a directory

Category: External Files
See: DOPTNUM Function in the documentation for your operating environment.

Syntax

DOPTNUM(directory-id)

Argument

directory-id
specifies the identifier that was assigned when the directory was opened, generally by the DOPEN function.

Restriction: The directory must have been previously opened by using DOPEN.

Details

Operating Environment Information: The number, names, and nature of the directory information varies between operating environments. The number of options that are available for a directory varies depending on the operating environment. For details, see the SAS documentation for your operating environment.

Examples

Example 1: Retrieving the Number of Information Items This example retrieves the number of system-dependent directory information items that are available for the directory MYDIR and closes the directory:

```
%let filrf=mydir;
%let rc=%sysfunc(filename(filrf,physical-name));
%let did=%sysfunc(dopen(&filrf));
%let infocnt=%sysfunc(doptnum(&did));
%let rc=%sysfunc(dclose(&did));
```

Example 2: Using DOPTNUM within a DATA Step This example creates a data set that retrieves the number of system-dependent information items that are available for the MYDIR directory:

```
data _null_;  
  rc=filename("mydir","physical-name");
  did=dopen("mydir");
  infocnt=doptnum(did);
```


\[
\begin{align*}
\text{rc} &= \text{dclose}(\text{did}); \\
\text{run};
\end{align*}
\]

**See Also**

Functions:

“DINFO Function” on page 499  
“DOPEN Function” on page 502  
“DOPTNAME Function” on page 503

---

**DREAD Function**

Returns the name of a directory member

**Category:** External Files

**Syntax**

\[
\text{DREAD}(\text{directory-id}, \text{nval})
\]

**Arguments**

- **directory-id**
  - Specifies the identifier that was assigned when the directory was opened, generally by the DOPEN function.
  - **Restriction:** The directory must have been previously opened by using DOPEN.

- **nval**
  - Specifies the sequence number of the member within the directory.

**Details**

DREAD returns a blank if an error occurs (such as when nval is out-of-range). Use DNUM to determine the highest possible member number that can be passed to DREAD.

**Examples**

This example opens the directory identified by the fileref MYDIR, retrieves the number of members, and places the number in the variable MEMCOUNT. It then retrieves the name of the last member, places the name in the variable LSTNAME, and closes the directory:

\[
\begin{align*}
\%let \text{filrf}=\text{mydir}; \\
\%let \text{rc}=%sysfunc(filename(\text{filrf}, \text{physical-name}));
\end{align*}
\]
%let did=%sysfunc(dopen(&filrf));
%let lstname=
%let memcount=%sysfunc(dnum(&did));
%if &memcount > 0 %then
   %let lstname=%sysfunc(dread(&did,&memcount));
%let rc=%sysfunc(dclose(&did));

See Also

Functions:
   “DNUM Function” on page 501
   “DOPEN Function” on page 502

DROPNOTE Function

Deletes a note marker from a SAS data set or an external file

Category:  SAS File I/O
Category:  External Files

Syntax

DROPNOTE(data-set-id|file-id,note-id)

Arguments

data-set-id|file-id
   specifies the identifier that was assigned when the data set or external file was
   opened, generally by the OPEN function or the FOPEN function.

note-id
   specifies the identifier that was assigned by the NOTE or FNOTE function.

Details

DROPNOTE deletes a marker set by NOTE or FNOTE. It returns a 0 if successful and
≠0 if not successful.

Examples

This example opens the SAS data set MYDATA, fetches the first observation, and
sets a note ID at the beginning of the data set. It uses POINT to return to the first
observation, and then uses DROPNOTE to delete the note ID:

   %let dsid=%sysfunc(open(mydata,i));
   %let rc=%sysfunc(fetch(&dsid));
   %let noteid=%sysfunc(note(&dsid));
   more macro statements
%let rc=%sysfunc(point(&dsid,&noteid));
%let rc=%sysfunc(fetch(&dsid));
%let rc=%sysfunc(dropnote(&dsid,&noteid));

See Also

Functions:
“FETCH Function” on page 522
“FNOTE Function” on page 547
“FOPEN Function” on page 549
“FPOINT Function” on page 554
“NOTE Function” on page 671
“OPEN Function” on page 690
“POINT Function” on page 715

DSNAME Function

Returns the SAS data set name that is associated with a data set identifier

**Category:** SAS File I/O

**Syntax**

\[
\text{DSNAME(data-set-id)}
\]

**Arguments**

*data-set-id*

specifies the data set identifier that the OPEN function returns.

**Details**

DSNAME returns the data set name that is associated with a data set identifier, or a blank if the data set identifier is not valid.

**Examples**

This example determines the name of the SAS data set that is associated with the variable DSID and displays the name in the SAS log.

%let dsid=%sysfunc(open(sasuser.houses,i));
%put The current open data set is %sysfunc(dsname(&dsid)).;
DUR Function

Returns the modified duration for an enumerated cash flow

Category: Financial

Syntax

\[ \text{DUR}(y,f,c(1), \ldots ,c(k)) \]

Arguments

\( y \)

specifies the effective per-period yield-to-maturity, expressed as a fraction.

Range: \( y > 0 \)

\( f \)

specifies the frequency of cash flows per period.

Range: \( f > 0 \)

\( c(1), \ldots ,c(k) \)

specifies a list of cash flows.

Details

The DUR function returns the value

\[
C = \frac{\sum_{k=1}^{K} k \left( \frac{c(k)}{(1 + fy)^k} \right)}{P (1 + y)}
\]

where

\[
P = \sum_{k=1}^{K} \frac{c(k)}{(1 + fy)^k}
\]

See Also

Function:

“OPEN Function” on page 690
Examples

```plaintext
data _null_; 
d=dur(1/20,1,.33,.44,.55,.49,.50,.22,.4,.8,.01,.36,.2,.4); 
put d; 
run; 
The value returned is 5.28402.
```

DURP Function

Returns the modified duration for a periodic cash flow stream, such as a bond.

Category: Financial

Syntax

```
DURP(A,c,n,K,k0,y)
```

Arguments

- **A**
  specifies the par value.
  **Range:** $A > 0$

- **c**
  specifies the nominal per-period coupon rate, expressed as a fraction.
  **Range:** $0 \leq c < 1$

- **n**
  specifies the number of coupons per period.
  **Range:** $n > 0$ and is an integer

- **K**
  specifies the number of remaining coupons.
  **Range:** $K > 0$ and is an integer

- **k₀**
  specifies the time from the present date to the first coupon date, expressed in terms of the number of periods.
  **Range:** $0 < k₀ \leq \frac{1}{n}$

- **y**
  specifies the nominal per-period yield-to-maturity, expressed as a fraction.
  **Range:** $y > 0$

Details

The DURP function returns the value
\[ D = \frac{1}{n} \sum_{k=1}^{K} \frac{t_k}{P(1 + \frac{v}{n})^t_k} \]

where

\[ t_k = nk_0 + k - 1 \]
\[ c(k) = \frac{c}{n} A \quad \text{for} \quad k = 1, \ldots, K - 1 \]
\[ c(K) = (1 + \frac{c}{n}) A \]

and where

\[ P = \sum_{k=1}^{K} \frac{c(k)}{(1 + \frac{v}{n})^{t_k}} \]

Examples

```plaintext
data _null_; 
d=durp(1000,1/100,4,14,.33/2,.10); 
put d; 
run;
```

The value returned is 3.26496.

ERF Function

Returns the value of the (normal) error function

**Category:** Mathematical

Syntax

**ERF**(argument)

Arguments

**argument**

is numeric.
Details

The ERF function returns the integral, given by

\[
\text{ERF}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-z^2} dz
\]

Examples

You can use the ERF function to find the probability (p) that a normally distributed random variable with mean 0 and standard deviation will take on a value less than X. For example, the quantity that is given by the following statement is equivalent to PROBNORM(X):

\[p = 0.5 + 0.5 \times \text{erf}(x/\sqrt{2})\]

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>y=erf(1.0);</td>
<td>0.8427007929</td>
</tr>
<tr>
<td>y=erf(-1.0);</td>
<td>-0.842700793</td>
</tr>
</tbody>
</table>

ERFC Function

Returns the value of the complementary (normal) error function

Category: Mathematical

Syntax

ERFC(argument)

Arguments

argument
- is numeric.

Details

The ERFC function returns the complement to the ERF function (that is, \(1 - \text{ERF}(\text{argument})\)).
### Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=erfc(1.0);</td>
<td>0.1572992071</td>
</tr>
<tr>
<td>x=erfc(-1.0);</td>
<td>.8427007929</td>
</tr>
</tbody>
</table>

### EUROCURR Function

**Converts one European currency to another**

**Category:** Currency Conversion

**See:** The EUROCURR function in *SAS National Language Support (NLS): User’s Guide*

### EXIST Function

**Verifies the existence of a SAS data library member**

**Category:** SAS File I/O

**Syntax**

\[
\text{EXIST}(\text{member-name}<,\text{member-type}<, \text{generation}>)
\]

**Arguments**

- **member-name**
  - specifies the SAS data library member. If `member-name` is blank or a null string, then EXIST uses the value of the `_LAST_` system variable as the `member-name`.

- **member-type**
  - specifies the type of SAS data library member. A few common member types are ACCESS, CATALOG, DATA, and VIEW. If you do not specify a `member-type`, then the member type DATA is assumed.

- **generation**
  - specifies the generation number of the SAS data set whose existence you are checking. If `member-type` is not DATA, `generation` is ignored.

  Positive numbers are absolute references to a historical version by its generation number. Negative numbers are relative references to a historical version in relation to the base version, from the youngest predecessor to the oldest. For example, -1 refers to the youngest version or, one version back from the base version. Zero is treated as a relative generation number.
Details

EXIST returns 1 if the library member exists, or 0 if member-name does not exist or member-type is invalid. Use CEXIST to verify the existence of an entry in a catalog.

Examples

Example 1: Verifying the Existence of a Data Set

This example verifies the existence of a data set. If the data set does not exist, then the example displays a message in the log:

```sas
%let dsname=sasuser.houses;
%macro opends(name);
%if %sysfunc(exist(&name)) %then
  %let dsid=%sysfunc(open(&name,i));
%else %put Data set &name does not exist.;
%mend opends;
%opends(&dsname);
```

Example 2: Verifying the Existence of a Data View

This example verifies the existence of the SAS data view TEST.MYVIEW. If the view does not exist, then the example displays a message in the log:

```sas
data _null_;
  dsname="test.myview";
  if (exist(dsname,"VIEW")) then
    dsid=open(dsname,"i");
  else put dsname 'does not exist.';
run;
```

Example 3: Determining If a Generation Data Set Exists

This example verifies the existence of a generation data set by using positive generation numbers (absolute reference):

```sas
data new(genmax=3);
  x=1;
run;
data new;
  x=99;
run;
data new;
  x=100;
run;
data new;
  x=101;
run;
data _null_;
  test=exist('new', 'DATA', 4);
  put test=
  test=exist('new', 'DATA', 3);
  put test=
  test=exist('new', 'DATA', 2);
  put test=
  test=exist('new', 'DATA', 1);
  put test=
run;
```
These lines are written to the SAS log:

```sas
    test=1
    test=1
    test=1
    test=0
```

You can change this example to verify the existence of the generation data set by using negative numbers (relative reference):

```sas
data new2(genmax=3);
    x=1;
run;
data new2;
    x=99;
run;
data new2;
    x=100;
run;
data new2;
    x=101;
run;
data _null_
    test=exist('new2', 'DATA', 0);
    put test=;
    test=exist('new2', 'DATA', -1);
    put test=;
    test=exist('new2', 'DATA', -2);
    put test=;
    test=exist('new2', 'DATA', -3);
    put test=;
    test=exist('new2', 'DATA', -4);
    put test=;
run;
```

These lines are written to the SAS log:

```sas
    test=0
    test=0
    test=0
    test=0
```

### See Also

Functions:

“CEXIST Function” on page 431
“FEXIST Function” on page 525
“FILEEXIST Function” on page 527

---

## EXP Function

Returns the value of the exponential function
FACT Function

Computes a factorial

Category: Mathematical

Syntax

FACT(n)

Arguments

n

is an integer that represents the number of elements for which the factorial is computed.

Details

The mathematical representation of the FACT function is given by the following equation:
\[ \text{FACT}(n) = n! \]

with \( n \geq 0 \).

If the expression cannot be computed, a missing value is returned.

**Examples**

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = \text{fact}(5) ; )</td>
<td>120</td>
</tr>
</tbody>
</table>

**See Also**

Functions:

- “COMB Function” on page 442
- “PERM Function” on page 714

---

**FAPPEND Function**

Appends the current record to the end of an external file

**Category:** External Files

**Syntax**

\[
\text{FAPPEND}(\text{file-id}<,\text{cc}>) \\
\]

**Arguments**

- **file-id**
  - specifies the identifier that was assigned when the file was opened, generally by the \text{FOPEN} function.

- **cc**
  - specifies a carriage control character:
    - \text{blank} indicates that the record starts a new line.
    - 0 skips one blank line before this new line.
    - - skips two blank lines before this new line.
    - 1 specifies that the line starts a new page.
specifies that the line overstrikes a previous line.

P specifies that the line is a terminal prompt.

= specifies that the line contains carriage control information.

all else specifies that the line record starts a new line.

Details

FAPPEND adds the record that is currently contained in the File Data Buffer (FDB) to the end of an external file. FAPPEND returns a 0 if the operation was successful and ≠0 if it was not successful.

Examples

This example assigns the fileref MYFILE to an external file and attempts to open the file. If the file is opened successfully, it moves data into the File Data Buffer, appends a record, and then closes the file. Note that in a macro statement you do not enclose character strings in quotation marks.

```%let filrf=myfile;
%let rc=%sysfunc(filename(filrf,
    physical-filename));
%let fid=%sysfunc(fopen(&filrf,a));
%if &fid > 0 %then
    %do;
    %let rc=%sysfunc(fput(&fid,
        Data for the new record));
    %let rc=%sysfunc(fappend(&fid));
    %let rc=%sysfunc(fclose(&fid));
    %end;
%else
    %do;
    /* unsuccessful open processing */
    %end;```

See Also

Functions:

“DOPEN Function” on page 502
“FCLOSE Function” on page 518
“FGET Function” on page 526
“FOPEN Function” on page 549
“FPUT Function” on page 556
“FWRITE Function” on page 563
“MOPEN Function” on page 656

FCLOSE Function

Closes an external file, directory, or directory member
Category: External Files
See: FCLOSE Function in the documentation for your operating environment.

Syntax
FCLOSE(file-id)

Argument

file-id
specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

Details
FCLOSE returns a 0 if the operation was successful and ≠0 if it was not successful. If you open a file within a DATA step, it is closed automatically when the DATA step ends.

Operating Environment Information: On some operating environments you must close the file with the FCLOSE function at the end of the DATA step. For details, see the SAS documentation for your operating environment.

Examples
This example assigns the fileref MYFILE to an external file, and attempts to open the file. If the file is opened successfully, indicated by a positive value in the variable FID, the program reads the first record, closes the file, and deassigns the fileref:

```sas
%let filrf=myfile;
%let rc=%sysfunc(filename(filrf,
   physical-filename));
%let fid=%sysfunc(fopen(&filrf));
%if &fid > 0 %then
   %do;
      %let rc=%sysfunc(fread(&fid));
      %let rc=%sysfunc(fclose(&fid));
   %end;
%else
   %do;
      %put %sysfunc(sysmsg());
   %end;
%let rc=%sysfunc(filename(filrf));
```

See Also

Functions:
“DCLOSE Function” on page 481
“DOPEN Function” on page 502
“FOPEN Function” on page 549
“FREAD Function” on page 558
FCOL Function

Returns the current column position in the File Data Buffer (FDB)

Category: External Files

Syntax

FCOL(file-id)

Argument

file-id

specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

Details

Use FCOL combined with FPOS to manipulate data in the File Data Buffer (FDB).

Examples

This example assigns the fileref MYFILE to an external file and attempts to open the file. If the file is successfully opened, indicated by a positive value in the variable FID, it puts more data into the FDB relative to position POS, writes the record, and closes the file:

```bash
%let filrf=myfile;
%let rc=%sysfunc(filename(filrf,
    physical-filename));
%let fid=%sysfunc(fopen(&filrf,o));
%if (&fid > 0) %then
  %do;
  %let record=This is data for the record.;
  %let rc=%sysfunc(fput(&fid,&record));
  %let pos=%sysfunc(fcol(&fid));
  %let rc=%sysfunc(fpos(&fid,%eval(&pos+1)));  
  %let rc=%sysfunc(fput(&fid,more data));
  %let rc=%sysfunc(fwrite(&fid));
  %let rc=%sysfunc(fclose(&fid));
  %end;
%let rc=%sysfunc(filename(filrf));
```

The new record written to the external file is

This is data for the record. more data
See Also

Functions:

“FCLOSE Function” on page 518
“FOPEN Function” on page 549
“FPOS Function” on page 555
“FPUT Function” on page 556
“FWRITE Function” on page 563
“MOPEN Function” on page 656

FDELETE Function

Deletes an external file or an empty directory

Category: External Files
See: FDELETE Function in the documentation for your operating environment.

Syntax

FDELETE(fileref | directory)

Argument

fileref

specifies the fileref that you assigned to the external file. You can assign filerefs by using the FILENAME statement or the FILENAME external file access function.

Restriction: The fileref that you use with FDELETE cannot be a concatenation.

Operating Environment Information: In some operating environments, you can specify a fileref that was assigned with an environment variable. You can also assign filerefs using system commands. For details, see the SAS documentation for your operating environment.

directory

specifies an empty directory that you want to delete.

Restriction: You must have authorization to delete the directory.

Details

FDELETE returns 0 if the operation was successful or ≠0 if it was not successful.
Examples

Example 1: Deleting an External File  This example generates a fileref for an external file in the variable FNAME. Then it calls FDELETE to delete the file and calls the FILENAME function again to deassign the fileref.

```sas
data _null_;  
  fname="tempfile";  
  rc=filename(fname,"physical-filename");  
  if rc = 0 and fexist(fname) then  
    rc=fdelete(fname);  
    rc=filename(fname);  
run;
```

Example 2: Deleting a Directory  This example uses FDELETE to delete an empty directory to which you have write access. If the directory is not empty, the optional SYSMSG function returns an error message stating that SAS is unable to delete the file.

```sas
filename testdir 'physical-filename';  
data _null_;  
rc=fdelete('testdir');  
put rc=;  
msg=sysmsg();  
put msg=;  
run;
```

See Also

Functions:
- “FEXIST Function” on page 525
- “FILENAME Function” on page 528

Statement:
- “FILENAME Statement” on page 1172

FETCH Function

Reads the next nondeleted observation from a SAS data set into the Data Set Data Vector (DDV)

Category:  SAS File I/O

Syntax

```
FETCH(data-set-id <,'NOSET'>)
```
Arguments

data-set-id
    specifies the data set identifier that the OPEN function returns.

'NOSET'
    prevents the automatic passing of SAS data set variable values to macro or DATA step variables even if the SET routine has been called.

Details

FETCH returns a 0 if the operation was successful, ≠0 if it was not successful, and –1 if the end of the data set is reached. FETCH skips observations marked for deletion.

If the SET routine has been called previously, the values for any data set variables are automatically passed from the DDV to the corresponding DATA step or macro variables. To override this behavior temporarily so that fetched values are not automatically copied to the DATA step or macro variables, use the NOSET option.

Examples

This example fetches the next observation from the SAS data set MYDATA. If the end of the data set is reached or if an error occurs, SYSMSG retrieves the appropriate message and writes it to the SAS log. Note that in a macro statement you do not enclose character strings in quotation marks.

```
%let dsid=%sysfunc(open(mydata,i));
%let rc=%sysfunc(fetch(&dsid));
%if &rc ne 0 %then
    %put %sysfunc(sysmsg());
%else
    %do;
    ...more macro statements...
    %end;
%let rc=%sysfunc(close(&dsid));
```

See Also

CALL Routine:
   “CALL SET Routine” on page 392

Functions:
   “FETCHOBS Function” on page 523
   “GETVARC Function” on page 571
   “GETVARN Function” on page 572

---

**FETCHOBS Function**

Reads a specified observation from a SAS data set into the Data Set Data Vector (DDV)

Category: SAS File I/O
Syntax

**FETCHOBS**(data-set-id,obs-number<,options>)

Arguments

data-set-id
specifies the data set identifier that the OPEN function returns.

obs-number
specifies the number of the observation to read. FETCHOBS treats the observation value as a relative observation number unless you specify the ABS option. The relative observation number may or may not coincide with the physical observation number on disk, because the function skips observations marked for deletion. When a WHERE clause is active, the function counts only observations that meet the WHERE condition.

Default: FETCHOBS skips deleted observations.

options
names one or more options, separated by blanks and enclosed in quotation marks:

'ABS'
specifies that the value of obs-number is absolute; that is, deleted observations are counted.

'NOSET'
prevents the automatic passing of SAS data set variable values to DATA step or macro variables even if the SET routine has been called.

Details

FETCHOBS returns 0 if the operation was successful, ≠0 if it was not successful, and −1 if the end of the data set is reached. To retrieve the error message that is associated with a non-zero return code, use the SYSMSG function. If the SET routine has been called previously, the values for any data set variables are automatically passed from the DDV to the corresponding DATA step or macro variables. To override this behavior temporarily, use the NOSET option.

If obs-number is less than 1, the function returns an error condition. If obs-number is greater than the number of observations in the SAS data set, the function returns an end-of-file condition.

Examples

This example fetches the tenth observation from the SAS data set MYDATA. If an error occurs, the SYSMSG function retrieves the error message and writes it to the SAS log. Note that in a macro statement you do not enclose character strings in quotation marks.

```sas
%let rc = %sysfunc(fetchobs(&mydataid,10));
%if &rc = -1 %then
   %put End of data set has been reached.;
%if &rc > 0 %then %put %sysfunc(sysmsg());
```
**FEXIST Function**

Verifies the existence of an external file associated with a fileref

**Category:** External Files

**See:** FEXIST Function in the documentation for your operating environment.

**Syntax**

\[
\text{FEXIST}(\text{fileref})
\]

**Argument**

\textit{fileref}

specifies the fileref assigned to an external file.

**Restriction:** The fileref must have been previously assigned.

**Operating Environment Information:** In some operating environments, you can specify a fileref that was assigned with an environment variable. For details, see the SAS documentation for your operating environment.

**Details**

FEXIST returns 1 if the external file that is associated with fileref exists, and 0 if the file does not exist. You can assign filerefs by using the FILENAME statement or the FILENAME external file access function. In some operating environments, you can also assign filerefs by using system commands.

**Comparison**

FILEEXIST verifies the existence of a file based on its physical name.

**Examples**

This example verifies the existence of an external file and writes the result to the SAS log:

\[
%\text{if}\%\text{sysfunc(fexist(\&fref))}\%\text{then}\\
\%\text{put}\%\text{The file identified by the fileref}
\]
&fref exists.;
%else
  %put %sysfunc(sysmsg());

See Also

Functions:
  “EXIST Function” on page 513
  “FILEEXIST Function” on page 527
  “FILENAME Function” on page 528
  “FILEREF Function” on page 531
Statement:
  “FILENAME Statement” on page 1172

---

**FGET Function**

Copies data from the File Data Buffer (FDB) into a variable

**Category:** External Files

**Syntax**

```lang
FGET(file-id,variable<,length>)
```

**Arguments**

- **file-id**
  - specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

- **variable**
  - in a DATA step, specifies a character variable to hold the data. In a macro, specifies a macro variable to hold the data. If `variable` is a macro variable and it does not exist, it is created.

- **length**
  - specifies the number of characters to retrieve from the FDB. If `length` is specified, only the specified number of characters is retrieved (or the number of characters remaining in the buffer if that number is less than `length`). If `length` is omitted, all characters in the FDB from the current column position to the next delimiter are returned. The default delimiter is a blank. The delimiter is not retrieved.

**See:** The “FSEP Function” on page 561 for more information about delimiters.

**Details**

FGET returns 0 if the operation was successful, or -1 if the end of the FDB was reached or no more tokens were available.
After FGET is executed, the column pointer moves to the next read position in the FDB.

**Examples**

This example assigns the fileref MYFILE to an external file and attempts to open the file. If the file is opened successfully, it reads the first record into the File Data Buffer, retrieves the first token of the record and stores it in the variable MYSTRING, and then closes the file. Note that in a macro statement you do not enclose character strings in quotation marks.

```sbasic
%let filrf=myfile;
%let rc=%sysfunc(filename(filrf,
  physical-filename));
%let fid=%sysfunc(fopen(&filrf));
%if &fid > 0 %then
  %do;
    %let rc=%sysfunc(fread(&fid));
    %let rc=%sysfunc(fget(&fid,mystring));
    %put &mystring;
    %let rc=%sysfunc(fclose(&fid));
  %end;
%let rc=%sysfunc(filename(filrf));
```

**See Also**

Functions:

- “FCLOSE Function” on page 518
- “FILENAME Function” on page 528
- “FOPEN Function” on page 549
- “FPOS Function” on page 555
- “FREAD Function” on page 558
- “FSEP Function” on page 561
- “MOPEN Function” on page 656

---

**FILEEXIST Function**

**Verifies the existence of an external file by its physical name**

**Category:** External Files

**See:** FILEEXIST Function in the documentation for your operating environment.

**Syntax**

```
FILEEXIST(file-name)
```
FILENAME Function

Assigns or deassigns a fileref to an external file, directory, or output device

Category: External Files

See: FILENAME Function in the documentation for your operating environment.

Syntax

FILENAME(fileref, file-name <,device-type <host-options <,dir-ref>>>)

Argument

file-name
specifies a fully qualified physical filename of the external file in the operating environment. In a DATA step, file-name can be a character expression, a string in quotation marks, or a DATA step variable. In a macro, file-name can be any expression.

Details

FILEEXIST returns 1 if the external file exists and 0 if the external file does not exist. The specification of the physical name for file-name varies according to the operating environment.

Although your operating environment utilities may recognize partial physical filenames, you must always use fully qualified physical filenames with FILEEXIST.

Examples

This example verifies the existence of an external file. If the file exists, FILEEXIST opens the file. If the file does not exist, FILEEXIST displays a message in the SAS log. Note that in a macro statement you do not enclose character strings in quotation marks.

%if %sysfunc(fileexist(&myfilerf)) %then
  %let fid=%sysfunc(fopen(&myfilerf));
%else
  %put The external file &myfilerf does not exist.;

See Also

Functions:
“EXIST Function” on page 513
“FEXIST Function” on page 525
“FILENAME Function” on page 528
“FILEREF Function” on page 531
“FOPEN Function” on page 549
Arguments

**fileref**
in a DATA step, specifies the fileref to assign to the external file. In a macro (for example, in the %SYSFUNC function), **fileref** is the name of a macro variable (without an ampersand) whose value contains the fileref to assign to the external file.

**Tip:** A blank **fileref** ("") causes an error. If the fileref is a DATA step character variable with a blank value and a minimum length of eight characters, or if a macro variable named in **fileref** has a null value, then a fileref is generated for you.

**file-name**
specifies the external file. Specifying a blank **file-name** deassigns one that was assigned previously.

**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 that are manipulated by the TEMP device can have the same attributes and behave identically to DISK files.

- **TERMINAL** specifies the user’s terminal.

- **UPRINTER** specifies a Universal Printing printer definition name.

**Operating Environment Information:** The FILENAME function also supports operating environment-specific devices. For details, see the SAS documentation for your operating environment.

**host-options**
specifies host-specific details such as file attributes and processing attributes. For details, see the SAS documentation for your operating environment.
dir-ref
specifies the fileref that was assigned to the directory or partitioned data set in which the external file resides.

Details
FILENAME returns 0 if the operation was successful; ≠0 if it was not successful. The name that is associated with the file or device is called a fileref (file reference name). Other system functions that manipulate external files and directories require that the files be identified by fileref rather than by physical filename.

Operating Environment Information: The term directory in this description refers to an aggregate grouping of files that are managed by the operating environment. Different operating environments identify these groupings with different names, such as directory, subdirectory, MACLIB, or partitioned data set. For details, see the SAS documentation for your operating environment.

Under some operating environments, you can also assign filerefs by using system commands. Depending on the operating environment, FILENAME might be unable to change or deassign filerefs that are assigned outside SAS.

The association between a fileref and a physical file lasts only for the duration of the current SAS session or until you change or discontinue the association by using FILENAME. You can deassign filerefs by specifying a null string for the file-name argument in FILENAME.

Examples

Example 1: Assigning a Fileref to an External File This example assigns the fileref MYFILE to an external file, then deassigns the fileref. Note that in a macro statement you do not enclose character strings in quotation marks.

```sas
%let filrf=myfile;
%let rc=%sysfunc(filename(filrf, physical-filename));
%if &rc ne 0 %then
  %put %sysfunc(sysmsg());
%let rc=%sysfunc(filename(filrf));
```

Example 2: Assigning a System-Generated Fileref This example assigns a system-generated fileref to an external file. The fileref is stored in the variable FNAME. Note that in a macro statement you do not enclose character strings in quotation marks.

```sas
%let rc=%sysfunc(filename(fname, physical-filename));
%if &rc %then
  %put %sysfunc(sysmsg());
%else
  %do;
    more macro statements
  %end;
```

Example 3: Assigning a Fileref to a Pipe File This example assigns the fileref MYPIPE to a pipe file with the output from the UNIX command LS, which lists the files in the directory /u/myid. Note that in a macro statement you do not enclose character strings in quotation marks.

```sas
%let filrf=mypipe;
%let rc=%sysfunc(filename(filrf, %str(ls /u/myid), pipe));
```
See Also

Functions:

“FEXIST Function” on page 525
“FILEEXIST Function” on page 527
“FILEREF Function” on page 531
“SYSMSG Function” on page 848

FILEREF Function

Verifies that a fileref has been assigned for the current SAS session

Category: External Files

See: FILEREF Function in the documentation for your operating environment.

Syntax

FILEREF(fileref)

Argument

fileref
specifies the fileref to be validated.

Range: 1 to 8 characters

Details

A negative return code indicates that the fileref exists but the physical file associated with the fileref does not exist. A positive value indicates that the fileref is not assigned. A value of zero indicates that the fileref and external file both exist.

A fileref can be assigned to an external file by using the FILENAME statement or the FILENAME function.

Operating Environment Information: Under some operating environments, filerefs can also be assigned by using system commands. For details, see the SAS documentation for your operating environment.

Examples

Example 1: Verifying that a Fileref is Assigned This example tests whether the fileref MYFILE is currently assigned to an external file. A system error message is issued if the fileref is not currently assigned:

```sas
%if %sysfunc(fileref(myfile))>0 %then
  %put MYFILE is not assigned;
```
Example 2: Verifying that Both a Fileref and a File Exist

This example tests for a zero value to determine if both the fileref and the file exist:

```%if %sysfunc(fileref(myfile)) ne 0 %then
  %put %sysfunc(sysmsg());```

See Also

Functions:

- “FEXIST Function” on page 525
- “FILEEXIST Function” on page 527
- “FILENAME Function” on page 528
- “SYSMSG Function” on page 848

Statement:
- “FILENAME Statement” on page 1172

---

**FIND Function**

**Searches for a specific substring of characters within a character string that you specify**

**Category:** Character

**Syntax**

FIND\(\text{string,substring},\text{modifiers}\)

FIND\(\text{string,substring},\text{startpos},\text{modifiers}\)

**Arguments**

**string**

specifies a character constant, variable, or expression that will be searched for substrings.

**Tip:** Enclose a literal string of characters in quotation marks.

**substring**

is a character constant, variable, or expression that specifies the substring of characters to search for in `string`.

**Tip:** Enclose a literal string of characters in quotation marks.

**modifiers**

is a character constant, variable, or expression that specifies one or more modifiers. The following `modifiers` can be in uppercase or lowercase:

- `i` ignores character case during the search. If this modifier is not specified, FIND only searches for character substrings with the same case as the characters in `substring`.

- `t` trims trailing blanks from `string` and `substring`.
Tip: If the modifier is a constant, enclose it in quotation marks. Specify multiple constants in a single set of quotation marks. Modifier can also be expressed as a variable or an expression that evaluates to one or more constants.

**startpos**

is an integer that specifies the position at which the search should start and the direction of the search.

**Details**

The FIND function searches string for the first occurrence of the specified substring, and returns the position of that substring. If the substring is not found in string, FIND returns a value of 0.

If startpos is not specified, FIND starts the search at the beginning of the string and searches the string from left to right. If startpos is specified, the absolute value of startpos determines the position at which to start the search. The sign of startpos determines the direction of the search.

<table>
<thead>
<tr>
<th>When startpos is ...</th>
<th>then FIND ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>greater than 0</td>
<td>starts the search at position startpos and the direction of the search is to the right. If startpos is greater than the length of string, FIND returns a value of 0.</td>
</tr>
<tr>
<td>less than 0</td>
<td>starts the search at position –startpos and the direction of the search is to the left. If –startpos is greater than the length of string, the search starts at the end of string.</td>
</tr>
<tr>
<td>equal to 0</td>
<td>returns a value of 0.</td>
</tr>
</tbody>
</table>

**Comparisons**

- The FIND function searches for substrings of characters in a character string, whereas the FINDC function searches for individual characters in a character string.
- The FIND function and the INDEX function both search for substrings of characters in a character string; however, the INDEX function does not have the modifiers nor the startpos arguments.

**Examples**

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>whereisshe=find(‘She sells seashells? Yes, she does.’,’she ’); put whereisshe;</td>
<td>27</td>
</tr>
<tr>
<td>variable1=’She sells seashells? Yes, she does.’; variable2=’she ’; variable3=’i’; whereisshe_i=find(variable1,variable2,variable3); put whereisshe_i;</td>
<td>1</td>
</tr>
<tr>
<td>expression1=’She sells seashells? ’</td>
<td></td>
</tr>
</tbody>
</table>
xyz='She sells seashells? Yes, she does.‘;
startposvar=22
whereisshe_22=find(xyz,'she',startposvar);
put whereisshe_22;

xyz='She sells seashells? Yes, she does.‘;
startposexp=1-23;
whereisShe_ineg22=find(xyz,'She','i',startposexp);
put whereisShe_ineg22;

See Also

Functions:
“COUNT Function” on page 466
“FINDC Function” on page 534
“INDEX Function” on page 588
“RXMATCH Function” on page 801

FINDC Function

Searches for specific characters that either appear or do not appear within a character string that you specify

Category: Character

Syntax

FINDC(string,characters<,modifiers><,startpos>)
FINDC(string,characters<,startpos><,modifiers>)

Arguments

string
specifies a character constant, variable, or expression that will be searched for characters.
Tip: Enclose a literal string of characters in quotation marks.

characters
is a character constant, variable, or expression that specifies one or more characters to search for in string.
Tip: Enclose a literal string of characters in quotation marks.

modifiers
is a character constant, variable, or expression that specifies one or more modifiers.
The following modifiers can be in uppercase or lowercase:
i ignores character case during the search. If this modifier is not specified, FINDC only searches for character substrings with the same case as the characters in characters.

o processes characters and modifiers only once, at the first call to this instance of FINDC. Consequently, if you change the value of characters or modifiers in subsequent calls, the change is ignored by FINDC.

t trims trailing blanks from string and characters.

v counts only the characters that do not appear in characters.

Tip: If the modifier is a constant, enclose it in quotation marks. Specify multiple constants in a single set of quotation marks. Modifier can also be expressed as a variable or an expression that evaluates to one or more constants.

startpos is an integer that specifies the position at which the search should start and the direction of the search.

Details

The FINDC function searches string for the first occurrence of the specified characters, and returns the position of the first character found. If the characters are not found in string, FINDC returns a value of 0.

If startpos is not specified, FINDC starts the search at the beginning of the string and searches the string from left to right. If startpos is specified, the absolute value of startpos determines the position at which to start the search. The sign of startpos determines the direction of the search.

<table>
<thead>
<tr>
<th>startpos is...</th>
<th>then FINDC...</th>
</tr>
</thead>
<tbody>
<tr>
<td>greater than 0</td>
<td>starts the search at position startpos and the direction of the search is to the right. If startpos is greater than the length of string, FINDC returns a value of 0.</td>
</tr>
<tr>
<td>less than 0</td>
<td>starts the search at position –startpos and the direction of the search is to the left. If –startpos is greater than the length of string, the search starts at the end of string.</td>
</tr>
<tr>
<td>equal to 0</td>
<td>returns a value of 0.</td>
</tr>
</tbody>
</table>

Comparisons

- The FINDC function searches for individual characters in a character string, whereas the FIND function searches for substrings of characters in a character string.
- The FINDC function and the INDEXC function both search for individual characters in a character string; however, the INDEXC function does not have the modifiers nor the startpos arguments.
- The FINDC function searches for individual characters in a character string, whereas the VERIFY function searches for the first character that is unique to an expression. The VERIFY function does not have the modifiers nor the startpos arguments.
Examples

Example 1: Searching for the Characters h, i, and Blank  
This example searches for the three characters h, i, and blank. The characters h and i are in lowercase. The uppercase characters H and I are ignored in this search.

data _null_;  
  whereishi=0;  
  do until(whereishi=0);  
    whereishi=findc('Hi there, Ian!', 'hi ', whereishi+1);  
    if whereishi=0 then put ‘The End’;  
    else do;  
      whatfound=substr('Hi there, Ian!', whereishi,1);  
      put whereishi= whatfound=;  
    end;  
  end;  
run;

The following lines are written to the SAS log:

whereishi=2 whatfound=i
whereishi=3 whatfound=
whereishi=5 whatfound=h
whereishi=10 whatfound=
The End

Example 2: Searching for the Characters h and i While Ignoring Case  
This example searches for the four characters h, i, H, and I. FINDC with the i modifier ignores character case during the search.

data _null_;  
  whereishi_i=0;  
  do until(whereishi_i=0);  
    variable1='Hi there, Ian!';  
    variable2='hi';  
    variable3='i';  
    whereishi_i=findc(variable1,variable2,variable3,whereishi_i+1);  
    if whereishi_i=0 then put ‘The End’;  
    else do;  
      whatfound=substr(variable1,whereishi_i,1);  
      put whereishi_i= whatfound=;  
    end;  
  end;  
run;

The following lines are written to the SAS log:

whereishi_i=1 whatfound=H
whereishi_i=2 whatfound=i
whereishi_i=5 whatfound=h
whereishi_i=11 whatfound=I
The End
Example 3: Searching for the Characters h and i with Trailing Blanks Trimmed  
This example searches for the two characters h and i. FINDC with the t modifier trims trailing blanks from the string argument and the characters argument.

data _null_
   ;
   whereishi_t=0;
   do until(whereishi_t=0);
      expression1='Hi there, '||'Ian!';
      expression2=kscan('bye or hi',3)||' ';
      expression3=trim('t ');
      whereishi_t=findc(expression1,expression2,expression3,whereishi_t+1);
      if whereishi_t=0 then put ''The End'';
      else do;
          whatfound=substr(expression1,whereishi_t,1);
          put whereishi_t= whatfound=;
      end;
   end;
run;

The following lines are written to the SAS log:

   whereishi_t=2 whatfound=i
   whereishi_t=5 whatfound=h
   The End

Example 4: Searching for all Characters, Excluding h, i, H, and I  
This example searches for all of the characters in the string, excluding the characters h, i, H, and I. FINDC with the v modifier counts only the characters that do not appear in the characters argument. This example also includes the i modifier and therefore ignores character case during the search.

data _null_
   ;
   whereishi_iv=0;
   do until(whereishi_iv=0);
      xyz='Hi there, Ian!';
      whereishi_iv=findc(xyz,'hi',whereishi_iv+1,'iv');
      if whereishi_iv=0 then put ''The End'';
      else do;
         whatfound=substr(xyz,whereishi_iv,1);
         put whereishi_iv= whatfound=;
      end;
   end;
run;

The following lines are written to the SAS log:

   whereishi_iv=3 whatfound=
   whereishi_iv=4 whatfound=t
   whereishi_iv=6 whatfound=e
   whereishi_iv=7 whatfound=r
   whereishi_iv=8 whatfound=e
   whereishi_iv=9 whatfound=,
The End

See Also

Functions:
“COUNTC Function” on page 468
“FIND Function” on page 532
“INDEXC Function” on page 589
“VERIFY Function” on page 883

FINFO Function

Returns the value of a file information item

Category: External Files
See: FINFO Function in the documentation for your operating environment.

Syntax
FINFO(file-id,info-item)

Arguments

file-id
specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

info-item
specifies the name of the file information item to be retrieved. This is a character value.

Details
FINFO returns the value of a system-dependent information item for an external file. FINFO returns a blank if the value given for info-item is invalid.

Operating Environment Information: The information available on files depends on the operating environment.

Comparisons

- FOPTNAME determines the names of the available file information items.
- FOPTNUM determines the number of system-dependent information items available.
Examples

This example stores information items about an external file in a SAS data set:

```sas
data info;
  length infoname infoval $60;
  drop rc fid infonum i close;
  rc=filename('abc','physical-filename');
  fid=fopen('abc');
  infonum=foptnum(fid);
  do i=1 to infonum;
    infoname=foptname(fid,i);
    infoval=finfo(fid,infoname);
    output;
  end;
  close=fclose(fid);
run;
```

See Also

Functions:
“FCLOSE Function” on page 518
“FOPTNUM Function” on page 553
“MOPEN Function” on page 656

FINV Function

Returns a quantile from the $F$ distribution

Category: Quantile

Syntax

FINV ($p$, $ndf$, $ddf$, $nc$)

Arguments

$p$

is a numeric probability.

Range: $0 \leq p < 1$

$ndf$

is a numeric numerator degrees of freedom parameter.

Range: $ndf > 0$

$ddf$

is a numeric denominator degrees of freedom parameter.

Range: $ddf > 0$
\( nc \)

is an optional numeric noncentrality parameter.

**Range:** \( nc \geq 0 \)

**Details**

The FINV function returns the \( p^{th} \) quantile from the \( F \) distribution with numerator degrees of freedom \( ndf \), denominator degrees of freedom \( ddf \), and noncentrality parameter \( nc \). The probability that an observation from the \( F \) distribution is less than the quantile is \( p \). This function accepts noninteger degrees of freedom parameters \( ndf \) and \( ddf \).

If the optional parameter \( nc \) is not specified or has the value 0, the quantile from the central \( F \) distribution is returned. The noncentrality parameter \( nc \) is defined such that if \( X \) and \( Y \) are normal random variables with means \( \mu \) and 0, respectively, and variance 1, then \( X^2 / Y^2 \) has a noncentral \( F \) distribution with \( nc = \mu^2 \).

**CAUTION:**
For large values of \( nc \), the algorithm could fail; in that case, a missing value is returned.

**Note:** FINV is the inverse of the PROBF function.

**Examples**

These statements compute the 95\(^{th} \) quantile value of a central \( F \) distribution with 2 and 10 degrees of freedom and a noncentral \( F \) distribution with 2 and 10.3 degrees of freedom and a noncentrality parameter equal to 2:

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>q1=finv(.95,2,10);</td>
<td>4.1028210151</td>
</tr>
<tr>
<td>q2=finv(.95,2,10.3,2);</td>
<td>7.583766024</td>
</tr>
</tbody>
</table>

**FIPNAME Function**

Converts two-digit FIPS codes to uppercase state names

**Category:** State and ZIP Code

**Syntax**

\[
\text{FIPNAME}(\text{expression})
\]

**Arguments**

\( expression \)

specifies a numeric expression that represents a U.S. FIPS code.
Details

If the FIPNAME function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 20.

The FIPNAME function converts a U.S. Federal Information Processing Standards (FIPS) code to the corresponding state or U.S. territory name in uppercase, returning a value of up to 20 characters.

Comparisons

The FIPNAME, FIPNAMEL, and FIPSTATE functions take the same argument but return different values. FIPNAME returns uppercase state names. FIPNAMEL returns mixed case state names. FIPSTATE returns a two-character state postal code (or world-wide GSA geographic code for U.S. territories) in uppercase.

Examples

The examples show the differences when using FIPNAME, FIPNAMEL, and FIPSTATE.

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=fipname(37); put x;</td>
<td>NORTH CAROLINA</td>
</tr>
<tr>
<td>x=fipnamel(37); put x;</td>
<td>North Carolina</td>
</tr>
<tr>
<td>x=fipstate(37); put x;</td>
<td>NC</td>
</tr>
</tbody>
</table>

See Also

Functions:

“FIPNAMEL Function” on page 541
“FIPSTATE Function” on page 542
“STFIPS Function” on page 832
“STNAME Function” on page 833
“STNAMEL Function” on page 834

FIPNAMEL Function

Converts two-digit FIPS codes to mixed case state names

Category: State and ZIP Code

Syntax

FIPNAMEL(expression)
Arguments

*expression*

specifies a numeric expression that represents a U.S. FIPS code.

Details

If the FIPNAMEL function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 20.

The FIPNAMEL function converts a U.S. Federal Information Processing Standards (FIPS) code to the corresponding state or U.S. territory name in mixed case, returning a value of up to 20 characters.

Comparisons

The FIPNAME, FIPNAMEL, and FIPSTATE functions take the same argument but return different values. FIPNAME returns uppercase state names. FIPNAMEL returns mixed case state names. FIPSTATE returns a two-character state postal code (or world-wide GSA geographic code for U.S. territories) in uppercase.

Examples

The examples show the differences when using FIPNAME, FIPNAMEL, and FIPSTATE.

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=fipname(37); put x;</td>
<td>NORTH CAROLINA</td>
</tr>
<tr>
<td>x=fipnamel(37); put x;</td>
<td>North Carolina</td>
</tr>
<tr>
<td>x=fipstate(37); put x;</td>
<td>NC</td>
</tr>
</tbody>
</table>

See Also

Functions:

“FIPNAME Function” on page 540
“FIPSTATE Function” on page 542
“STFIPS Function” on page 832
“STNAME Function” on page 833
“STNAMEL Function” on page 834

FIPSTATE Function

Converts two-digit FIPS codes to two-character state postal codes
Category: State and ZIP Code

Syntax

\[ \text{FIPSTATE}(\text{expression}) \]

Arguments

\textit{expression}

specifies a numeric expression that represents a U.S. FIPS code.

Details

If the FIPSTATE function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 20.

The FIPSTATE function converts a U.S. Federal Information Processing Standards (FIPS) code to a two-character state postal code (or world-wide GSA geographic code for U.S. territories) in uppercase.

Comparisons

The FIPNAME, FIPNAMES, and FIPSTATE functions take the same argument but return different values. FIPNAME returns uppercase state names. FIPNAMES returns mixed case state names. FIPSTATE returns a two-character state postal code (or world-wide GSA geographic code for U.S. territories) in uppercase.

Examples

The examples show the differences when using FIPNAME, FIPNAMES, and FIPSTATE.

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=fipname(37);</td>
<td>NORTH CAROLINA</td>
</tr>
<tr>
<td>put x;</td>
<td></td>
</tr>
<tr>
<td>x=fipname(37);</td>
<td></td>
</tr>
<tr>
<td>put x;</td>
<td>North Carolina</td>
</tr>
<tr>
<td>x=fipstate(37);</td>
<td></td>
</tr>
<tr>
<td>put x;</td>
<td>NC</td>
</tr>
</tbody>
</table>

See Also

Functions:

“FIPNAME Function” on page 540
“FIPNAMES Function” on page 541
“STFIPS Function” on page 832
“STNAME Function” on page 833
FLOOR Function

Returns the largest integer that is less than or equal to the argument, fuzzed to avoid unexpected floating-point results

Category: Truncation

Syntax

FLOOR (argument)

Arguments

argument is numeric.

Details

If the argument is within 1E-12 of an integer, the function returns that integer.

Comparisons

Unlike the FLOORZ function, the FLOOR function fuzzes the result. If the argument is within 1E-12 of an integer, the FLOOR function fuzzes the result to be equal to that integer. The FLOORZ function does not fuzz the result. Therefore, with the FLOORZ function you might get unexpected results.

Examples

The following SAS statements produce these results.

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>var1=2.1; a=floor(var1); put a;</td>
<td>2</td>
</tr>
<tr>
<td>var2=-2.4; b=floor(var2); put b;</td>
<td>-3</td>
</tr>
<tr>
<td>c=floor(-1.6); put c;</td>
<td>-2</td>
</tr>
<tr>
<td>d=floor(1.-1.e-13); put d;</td>
<td>1</td>
</tr>
</tbody>
</table>
SAS Statements | Results
--- | ---
e=floor(763);
put e; | 763
f=floor(-223.456);
put f; | -224

See Also

Functions:
“FLOORZ Function” on page 545

FLOORZ Function

Returns the largest integer that is less than or equal to the argument, using zero fuzzing

Category: Truncation

Syntax

FLOORZ \((\text{argument})\)

Arguments

\(\text{argument}\)

is a numeric constant, variable, or expression.

Comparisons

Unlike the FLOOR function, the FLOORZ function uses zero fuzzing. If the argument is within 1E-12 of an integer, the FLOOR function fuzzes the result to be equal to that integer. The FLOORZ function does not fuzz the result. Therefore, with the FLOORZ function you might get unexpected results.

Examples

The following SAS statements produce these results.

SAS Statements | Results
--- | ---
\(\text{var1}=2.1;\)
\(\text{a=floorz(var1)};\)
\(\text{put a;}\) | 2
\(\text{var2}=-2.4;\)
\(\text{b=floorz(var2)};\)
\(\text{put b;}\) | -3
c=floorz(-1.6);
put c; | -2
SAS Statements Results

\[
\begin{array}{|l|}
\hline
\text{var6=1.-1.e-13;} \\
\text{d=floorz(1.-1.e-13);} \\
\text{put d;} \\
\text{e=floorz(763);} \\
\text{put e;} \\
\text{f=floorz(-223.456);} \\
\text{put f;} \\
\hline
0 \\
763 \\
-224 \\
\hline
\end{array}
\]

See Also

Functions:

“FLOOR Function” on page 544

FNONCT Function

Returns the value of the noncentrality parameter of an F distribution

Category: Mathematical

Syntax

\[
\text{FNONCT}(x, ndf, ddf, prob)
\]

Arguments

\[
\begin{aligned}
\text{x} & \quad \text{is a numeric random variable.} \\
& \quad \text{Range: } x \geq 0 \\
\text{ndf} & \quad \text{is a numeric numerator degrees-of-freedom parameter.} \\
& \quad \text{Range: } ndf > 0 \\
\text{ddf} & \quad \text{is a numeric denominator degrees-of-freedom parameter.} \\
& \quad \text{Range: } ddf > 0 \\
\text{prob} & \quad \text{is a probability.} \\
& \quad \text{Range: } 0 < prob < 1
\end{aligned}
\]

Details

The FNONCT function returns the nonnegative noncentrality parameter from a noncentral F distribution whose parameters are \( x, \text{ndf}, \text{ddf}, \) and \( nc \). If \( prob \) is greater
than the probability from the central $F$ distribution whose parameters are $x$, $ndf$, and $ddf$, a root to this problem does not exist. In this case a missing value is returned. A Newton-type algorithm is used to find a nonnegative root $nc$ of the equation

$$P_f (x|ndf, ddf, nc) - prob = 0$$

where

$$P_f (x|ndf, ddf, nc) = e^{-\frac{nc}{2}} \sum_{j=0}^{\infty} \frac{(nc)^j}{j!} I_{\frac{(ndf)x}{ddf+(ndf)x}} \left(\frac{ddf}{2} + j, \frac{ddf}{2}\right)$$

where $I(\ldots)$ is the probability from the beta distribution that is given by

$$I_x(a, b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)} \int_{0}^{x} t^{a-1}(1-t)^{b-1} dt$$

If the algorithm fails to converge to a fixed point, a missing value is returned.

**Examples**

```plaintext
data work;
x=2;
df=4;
ddf=5;
do nc=1 to 3 by .5;
   prob=probf(x,df,ddf,nc);
   ncc=fnonct(x,df,ddf,prob);
   output;
end;
run;
proc print;
run;
```

**Output 4.24  FNONCT Example Output**

<table>
<thead>
<tr>
<th>OBS</th>
<th>x</th>
<th>df</th>
<th>ddf</th>
<th>nc</th>
<th>prob</th>
<th>ncc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>1.0</td>
<td>0.69277</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>1.5</td>
<td>0.65701</td>
<td>1.5</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>2.0</td>
<td>0.62232</td>
<td>2.0</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>2.5</td>
<td>0.58878</td>
<td>2.5</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>3.0</td>
<td>0.55642</td>
<td>3.0</td>
</tr>
</tbody>
</table>

**FNOTE Function**

Identifies the last record that was read and returns a value that FPOINT can use.
Category: External Files

Syntax

\texttt{FNOTE(file-id)}

Argument

\textit{file-id}

specifies the identifier that was assigned when the file was opened, generally by the \texttt{FOPEN} function.

Details

You can use \texttt{FNOTE} like a bookmark, marking the position in the file so that your application can later return to that position using \texttt{FPOINT}. The value returned by \texttt{FNOTE} is required by the \texttt{FPOINT} function to reposition the file pointer on a specific record.

To free the memory associated with each \texttt{FNOTE} identifier, use \texttt{DROPNOTE}.

Examples

This example assigns the fileref \texttt{MYFILE} to an external file and attempts to open the file. If the file is opened successfully, indicated by a positive value in the variable \texttt{FID}, then it reads the records, stores in the variable \texttt{NOTE3} the position of the third record read, and then later uses \texttt{FPOINT} to point back to \texttt{NOTE3} to update the file. After updating the record, it closes the file:

```plaintext
%let fref=MYFILE;
%let rc=%sysfunc(filename(fref,
   physical-filename));
%let fid=%sysfunc(fopen(&fref,u));
%if &fid > 0 %then 
  %do;
    %let rc=%sysfunc(fread(&fid)); /* Read second record. */
    %let rc=%sysfunc(fread(&fid)); /* Read third record. */
    %let rc=%sysfunc(fread(&fid)); /* Note position of third record. */
    %let note3=%sysfunc(fnote(&fid)); /* Read fourth record. */
    %let rc=%sysfunc(fread(&fid)); /* Read fifth record. */
    %let rc=%sysfunc(fread(&fid)); /* Point to third record. */
    %let rc=%sysfunc(fpoint(&fid,&note3)); /* Read third record. */
    %let rc=%sysfunc(fread(&fid)); /* Copy new text to FDB. */
    %let rc=%sysfunc(fput(&fid,New text));
  %end;
```
/* Update third record */
/* with data in FDB. */
%let rc=%sysfunc(fwrite(&fid));
/* Close file. */
%let rc=%sysfunc(fclose(&fid));
%end;
%let rc=%sysfunc(filename(fref));

See Also

Functions:
“DROPNOTE Function” on page 507
“FCLOSE Function” on page 518
“FILENAME Function” on page 528
“FOPEN Function” on page 549
“FPOINT Function” on page 554
“FPUT Function” on page 556
“FREAD Function” on page 558
“FREWIND Function” on page 559
“FWRITE Function” on page 563
“MOPEN Function” on page 656

FOPEN Function

Opens an external file and returns a file identifier value

Category:  External Files
See:  FOPEN Function in the documentation for your operating environment.

Syntax

FOPEN(fileref<,open-mode<,record-length<,record-format>>>)

Arguments

fileref
specifies the fileref assigned to the external file.

open-mode
specifies the type of access to the file:

A  APPEND mode allows writing new records after the current end of the file.
I  INPUT mode allows reading only (default).
O  OUTPUT mode defaults to the OPEN mode specified in the operating environment option in the FILENAME statement or
function. If no operating environment option is specified, it allows writing new records at the beginning of the file.

S  Sequential input mode is used for pipes and other sequential devices such as hardware ports.

U  UPDATE mode allows both reading and writing.

Default:  I

record-length
specifies the logical record length of the file. To use the existing record length for the file, specify a length of 0, or do not provide a value here.

record-format
specifies the record format of the file. To use the existing record format, do not specify a value here. Valid values are:

B  data are to be interpreted as binary data.
D  use default record format.
E  use editable record format.
F  file contains fixed length records.
P  file contains printer carriage control in operating environment-dependent record format. Note: For z/OS data sets with FBA or VBA record format, specify ‘P’ for the record-format argument.
V  file contains variable length records.

Details

CAUTION:
Use OUTPUT mode with care. Opening an existing file for output overwrites the current contents of the file without warning.

The FOPEN function opens an external file for reading or updating and returns a file identifier value that is used to identify the open file to other functions. You must associate a fileref with the external file before calling the FOPEN function. FOPEN returns a 0 if the file could not be opened. You can assign filerefs by using the FILENAME statement or the FILENAME external file access function. Under some operating environments, you can also assign filerefs by using system commands.

Operating Environment Information:  On some operating environments you must close the file at the end of the DATA step using the FCLOSE function. For details, see the SAS documentation for your operating environment.

Examples

Example 1: Opening a File Using Defaults  This example assigns the fileref MYFILE to an external file and attempts to open the file for input using all defaults. Note that in a macro statement you do not enclose character strings in quotation marks.

```
%let filrf=myfile;
%let rc=%sysfunc(filename(filrf,
    physical-filename));
%let fid=%sysfunc(fopen(&filrf));
```
Example 2: Opening a File without Using Defaults  This example attempts to open a file for input without using defaults. Note that in a macro statement you do not enclose character strings in quotation marks.

%let fid=%sysfunc(fopen(file2,o,132,e));

See Also

Functions:

“DOPEN Function” on page 502
“FCLOSE Function” on page 518
“FILENAME Function” on page 528
“FILEREF Function” on page 531
“MOPEN Function” on page 656

Statement:

“FILENAME Statement” on page 1172

FOPTNAME Function

Returns the name of an item of information about a file

Category:  External Files

See:  FOPTNAME Function in the documentation for your operating environment.

Syntax

FOPTNAME(file-id,nval)

Arguments

file-id
specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

nval
specifies the number of the information item.

Details

FOPTNAME returns a blank if an error occurred.

Operating Environment Information:  The number, value, and type of information items that are available depend on the operating environment.  △
Examples

Example 1: Retrieving File Information Items and Writing Them to the Log  
This example retrieves the system-dependent file information items that are available and writes them to the log:

```sas
%let filrf=myfile;
%let rc=%sysfunc(filename(filrf,
    physical-filename));
%let fid=%sysfunc(fopen(&filrf));
%let infonum=%sysfunc(foptnum(&fid));
%do j=1 %to &infonum;
    %let name=%sysfunc(foptname(&fid,&j));
    %let value=%sysfunc(finfo(&fid,&name));
    %put File attribute &name equals &value;
%end;
%let rc=%sysfunc(fclose(&fid));
%let rc=%sysfunc(filename(filrf));
```

Example 2: Creating a Data Set with Names and Values of File Attributes  
This example creates a data set that contains the name and value of the available file attributes:

```sas
data fileatt;
    length name $ 20 value $ 40;
    drop rc fid j infonum;
    rc=filename("myfile","physical-filename");
    fid=fopen("myfile");
    infonum=foptnum(fid);
    do j=1 to infonum;
        name=foptname(fid,j);
        value=finfo(fid,name);
        put 'File attribute ' name
            'has a value of ' value;
        output;
    end;
end;
rc=filename("myfile");
run;
```

See Also

Functions:

- “DINFO Function” on page 499
- “DOPTNAME Function” on page 503
- “DOPTNUM Function” on page 505
- “FCLOSE Function” on page 518
- “FILENAME Function” on page 528
- “FINFO Function” on page 538
- “FOPEN Function” on page 549
- “FOPTNUM Function” on page 553
- “MOPEN Function” on page 656
FOPTNUM Function

Returns the number of information items that are available for an external file

Category: External Files

See: FOPTNUM Function in the documentation for your operating environment.

Syntax

FOPTNUM(file-id)

Argument

file-id

specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

Details

Operating Environment Information: The number, value, and type of information items that are available depend on the operating environment.

Comparisons

□ Use FOPTNAME to determine the names of the items that are available for a particular operating environment.
□ Use FINFO to retrieve the value of a particular information item.

Examples

This example opens the external file with the fileref MYFILE and determines the number of system-dependent file information items available:

```
%let fid=%sysfunc(fopen(myfile));
%let infonum=%sysfunc(foptnum(&fid));
```

See Also

Functions:

“DINFO Function” on page 499
“DOPTNAME Function” on page 503
“DOPTNUM Function” on page 505
“FINFO Function” on page 538
“FOPEN Function” on page 549
“FOPTNAME Function” on page 551
“MOPEN Function” on page 656

See the “Examples” on page 552 in the FOPTNAME Function.
FPOINT Function

Positions the read pointer on the next record to be read

Category: External Files

Syntax

FPOINT(file-id,note-id)

Arguments

file-id

specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

note-id

specifies the identifier that was assigned by the FNOTE function.

Details

FPOINT returns 0 if the operation was successful, or ≠0 if it was not successful.

FPOINT determines only the record to read next. It has no impact on which record is written next. When you open the file for update, FWRITE writes to the most recently read record.

Examples

This example assigns the fileref MYFILE to an external file and attempts to open the file. If the file is opened successfully, it reads the records and uses NOTE3 to store the position of the third record read. Later, it points back to NOTE3 to update the file, and closes the file afterward:

```plaintext
%let filrf=myfile;
%let rc=%sysfunc(filename(filrf,
  physical-filename));
%let fid=%sysfunc(fopen(&filrf,u));
%if &fid > 0 %then
  %do;
    /* Read first record. */
    %let rc=%sysfunc(fread(&fid));
    /* Read second record. */
    %let rc=%sysfunc(fread(&fid));
    /* Read third record. */
    %let rc=%sysfunc(fread(&fid));
    /* Note position of third record. */
    %let note3=%sysfunc(fnote(&fid));
    /* Read fourth record. */
    %let rc=%sysfunc(fread(&fid));
    /* Read fifth record. */
    %let rc=%sysfunc(fread(&fid));
  %end;
```

See Also

Functions:
“DROPNOTE Function” on page 507
“FCLOSE Function” on page 518
“FILENAME Function” on page 528
“FNOTE Function” on page 547
“FOPEN Function” on page 549
“FPUT Function” on page 556
“FREAD Function” on page 558
“FREWIND Function” on page 559
“FWRITE Function” on page 563
“MOPEN Function” on page 656

FPOS Function

Sets the position of the column pointer in the File Data Buffer (FDB)

Category: External Files

Syntax
FPOS(file-id,nval)

Arguments

file-id
specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

nval
specifies the column at which to set the pointer.
Details

FPOS returns 0 if the operation was successful, ≠0 if it was not successful. If the specified position is past the end of the current record, the size of the record is increased appropriately. However, in a fixed block or VBA file, if you specify a column position beyond the end of the record, the record size does not change and the text string is not written to the file.

Examples

This example assigns the fileref MYFILE to an external file and attempts to open the file. If the file is opened successfully, indicated by a positive value in the variable FID, it places data into the file’s buffer at column 12, writes the record, and closes the file:

```sas
%let filrf=myfile;
%let rc=%sysfunc(filename(filrf,
    physical-filename));
%let fid=%sysfunc(fopen(&filrf,o));
%if (&fid > 0) %then
  %do;
    %let dataline=This is some data.;
    /* Position at column 12 in the FDB. */
    %let rc=%sysfunc(fpos(&fid,12));
    /* Put the data in the FDB. */
    %let rc=%sysfunc(fput(&fid,&dataline));
    /* Write the record. */
    %let rc=%sysfunc(fwrite(&fid));
    /* Close the file. */
    %let rc=%sysfunc(fclose(&fid));
  %end;
%let rc=%sysfunc(filename(filrf));
```

See Also

Functions:

“FCLOSE Function” on page 518
“FCOL Function” on page 520
“FILENAME Function” on page 528
“FOPEN Function” on page 549
“FPUT Function” on page 556
“FWRITE Function” on page 563
“MOPEN Function” on page 656

FPUT Function

Moves data to the File Data Buffer (FDB) of an external file, starting at the FDB’s current column position

Category: External Files
Functions and CALL Routines

FPUT Function 557

Syntax

\[ \text{FPUT}(\text{file-id}, \text{cval}) \]

Arguments

\textit{file-id}

specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

\textit{cval}

specifies the file data. In a DATA step, \textit{cval} can be a character string in quotation marks or a DATA step variable. In a macro, \textit{cval} is a macro variable.

Details

FPUT returns 0 if the operation was successful, ≠0 if it was not successful. The number of bytes moved to the FDB is determined by the length of the variable. The value of the column pointer is then increased to one position past the end of the new text.

Examples

This example assigns the fileref MYFILE to an external file and attempts to open the file in APPEND mode. If the file is opened successfully, indicated by a positive value in the variable FID, it moves data to the FDB using FPUT, appends a record using FWRITE, and then closes the file. Note that in a macro statement you do not enclose character strings in quotation marks.

\[
\begin{align*}
% \text{let filrf=myfile;} \\
% \text{let rc=%sysfunc(filename(filrf, physical-filename));} \\
% \text{let fid=%sysfunc(fopen(&filrf,a));} \\
% \text{if &fid > 0 \%then \%do;} \\
% \text{\%let mystring=This is some data.;} \\
% \text{\%let rc=%sysfunc(fput(&fid,&mystring));} \\
% \text{\%let rc=%sysfunc(fwrite(&fid));} \\
% \text{\%let rc=%sysfunc(fclose(&fid));} \\
% \text{\%end;} \\
% \text{\%else} \\
% \text{\%put %sysfunc(sysmsg());} \\
% \text{\%let rc=%sysfunc(filename(filrf));}
\end{align*}
\]

See Also

Functions:

“FCLOSE Function” on page 518
“FILENAME Function” on page 528
“FNOTE Function” on page 547
“FOPEN Function” on page 549
“FPOINT Function” on page 554
“FPOS Function” on page 555
FREAD Function

Reads a record from an external file into the File Data Buffer (FDB)

Category: External Files

Syntax

FREAD(file-id)

Argument

file-id
specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

Details

FREAD returns 0 if the operation was successful, ≠0 if it was not successful. The position of the file pointer is updated automatically after the read operation so that successive FREAD functions read successive file records.

To position the file pointer explicitly, use FNOTE, FPOINT, and FREWIND.

Examples

This example assigns the fileref MYFILE to an external file and attempts to open the file. If the file opens successfully, it lists all of the file's records in the log:

```plaintext
%let filrf=myfile;
%let rc=%sysfunc(filename(filrf,
  physical-filename));
%let fid=%sysfunc(fopen(&filrf));
%if &fid > 0 %then
  %do %while(%sysfunc(fread(&fid)) = 0);
    %let rc=%sysfunc(fget(&fid,c,200));
    %put &c;
  %end;
%let rc=%sysfunc(fclose(&fid));
%let rc=%sysfunc(filename(filrf));
```

See Also

Functions:

“FWRITE Function” on page 563
“MOPEN Function” on page 656
“SYSMSG Function” on page 848
FREWIND Function

Positions the file pointer to the start of the file

Category: External Files

Syntax

FREWIND(file-id)

Argument

file-id
  specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

Details

FREWIND returns 0 if the operation was successful, ≠0 if it was not successful. FREWIND has no effect on a file opened with sequential access.

Examples

This example assigns the fileref MYFILE to an external file. Then it opens the file and reads the records until the end of the file is reached. The FREWIND function then repositions the pointer to the beginning of the file. The first record is read again and stored in the File Data Buffer (FDB). The first token is retrieved and stored in the macro variable VAL:

```%let filrf=myfile;
%let rc=%sysfunc(filename(filrf,
  physical-filename));
%let fid=%sysfunc(fopen(&filrf));
%let rc=0;
%do %while (&rc ne -1);
  /* Read a record. */
  %let rc=%sysfunc(fread(&fid));
%end;```
/* Reposition pointer to beginning of file. */
%if &rc = -1 %then
  %do;
  %let rc=%sysfunc(frewind(&fid));
  /* Read first record. */
  %let rc=%sysfunc(fread(&fid));
  /* Read first token */
  /* into macro variable VAL. */
  %let rc=%sysfunc(fget(&fid,val));
  %put val=&val;
  %end;
%else
  %put Error on fread=%sysfunc(sysmsg());
  %let rc=%sysfunc(fclose(&fid));
  %let rc=%sysfunc(filename(filrf));
%end;

See Also

Functions:
“FCLOSE Function” on page 518
“FGET Function” on page 526
“FILENAME Function” on page 528
“FOPEN Function” on page 549
“FREAD Function” on page 558
“MOPEN Function” on page 656
“SYMSG Function” on page 848

FRLEN Function

Returns the size of the last record read, or, if the file is opened for output, returns the current record size

Category: External Files

Syntax
FRLEN(file-id)

Argument

file-id
specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.
Examples

This example opens the file that is identified by the fileref MYFILE. It determines the minimum and maximum length of records in the external file and writes the results to the log:

```sas
%let fid=%sysfunc(fopen(myfile));
%let min=0;
%let max=0;
%if (%sysfunc(fread(&fid)) = 0) %then
  %do;
    %let min=%sysfunc(frlen(&fid));
    %let max=&min;
  %do %while(%sysfunc(fread(&fid)) = 0);
    %let reclen=%sysfunc(frlen(&fid));
    %if (&reclen > &max) %then
      %let max=&reclen;
    %if (&reclen < &min) %then
      %let min=&reclen;
  %end;
%end;
%let rc=%sysfunc(fclose(&fid));
%put max=&max min=&min;
```

See Also

Functions:
- “FCLOSE Function” on page 518
- “FOPEN Function” on page 549
- “FREAD Function” on page 558
- “MOPEN Function” on page 656

FSEP Function

Sets the token delimiters for the FGET function

Category: External Files

Syntax

FSEP(file-id,character(s))

Arguments

file-id

specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.
**character**

specifies one or more delimiters that separate items in the File Data Buffer (FDB). Each character listed is a delimiter. That is, if *character* is `#@`, either `#` or `@` can separate items. Multiple consecutive delimiters, such as `@#@`, are treated as a single delimiter.

**Default:** blank

**Details**

FSEP returns 0 if the operation was successful, ≠0 if it was not successful.

**Examples**

An external file has data in this form:

John J. Doe, Male, 25, Weight Lifter
Pat O’Neal, Female, 22, Gymnast

Note that each field is separated by a comma.

This example reads the file that is identified by the fileref MYFILE, using the comma as a separator, and writes the values for NAME, GENDER, AGE, and WORK to the SAS log. Note that in a macro statement you do not enclose character strings in quotation marks.

```sas
%let fid=%sysfunc(fopen(myfile));
%let rc=%sysfunc(fsep(&fid,%str(,)));  
%do %while(%sysfunc(fread(&fid)) = 0);
  %let rc=%sysfunc(fget(&fid,name));
  %let rc=%sysfunc(fget(&fid,gender));
  %let rc=%sysfunc(fget(&fid,age));
  %let rc=%sysfunc(fget(&fid,work));
  %put name=&bquote(&name) gender=&gender 
     age=&age work=&work;
%end;
%let rc=%sysfunc(fclose(&fid));
```

**See Also**

Functions:

“FCLOSE Function” on page 518
“FGET Function” on page 526
“FOPEN Function” on page 549
“FREAD Function” on page 558
“MOPEN Function” on page 656

---

**FUZZ Function**

Returns the nearest integer if the argument is within **1E-12**

**Category:** Truncation
Syntax

FUZZ(argument)

Arguments

argument

is numeric.

Details

The FUZZ function returns the nearest integer value if the argument is within 1E−12 of the integer (that is, if the absolute difference between the integer and argument is less than 1E−12). Otherwise, the argument is returned.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>var1=5.999999999999999;</td>
<td></td>
</tr>
<tr>
<td>x=fuzz(var1);</td>
<td></td>
</tr>
<tr>
<td>put x 16.14;</td>
<td>6.000000000000000</td>
</tr>
<tr>
<td>x=fuzz(5.99999999);</td>
<td></td>
</tr>
<tr>
<td>put x 16.14;</td>
<td>5.999999990000000</td>
</tr>
</tbody>
</table>

FWRITE Function

Writes a record to an external file

Category: External Files

Syntax

FWRITE(file-id,<.cc>)

Arguments

file-id

specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.
cc specifies a carriage-control character:

- blank starts the record on a new line.
- 0 skips one blank line before a new line.
- - skips two blank lines before a new line.
- 1 starts the line on a new page.
- + overstrikes the line on a previous line.
- P interprets the line as a terminal prompt.
- = interprets the line as carriage control information.
- all else starts the line record on a new line.

Details

FWRITE returns 0 if the operation was successful, ≠0 if it was not successful. FWRITE moves text from the File Data Buffer (FDB) to the external file. In order to use the carriage control characters, you must open the file with a record format of P (print format) in FOPEN.

Examples

This example assigns the fileref MYFILE to an external file and attempts to open the file. If the file is opened successfully, it writes the numbers 1 to 50 to the external file, skipping two blank lines. Note that in a macro statement you do not enclose character strings in quotation marks.

```%let filrf=myfile;
%let rc=%sysfunc(filename(filrf,
   physical-filename));
%let fid=%sysfunc(fopen(&filrf,o,0,P));

%do i=1 %to 50;
  %let rc=%sysfunc(fput(&fid,
     %sysfunc(putn(&i,2.))));

  %if (%sysfunc(fwrite(&fid,-)) ne 0) %then
    %put %sysfunc(sysmsg());
%end;

%let rc=%sysfunc(fclose(&fid));
```

See Also

Functions:

- “FAPPEND Function” on page 517
- “FCLOSE Function” on page 518
- “FGET Function” on page 526
- “FILENAME Function” on page 528
- “FOPEN Function” on page 549
GAMINV Function

Returns a quantile from the gamma distribution
Category: Quantile

Syntax
GAMINV(p,a)

Arguments

p is a numeric probability.
Range: 0 ≤ p < 1

a is a numeric shape parameter.
Range: a > 0

Details
The GAMINV function returns the \( p \)th quantile from the gamma distribution, with shape parameter \( a \). The probability that an observation from a gamma distribution is less than or equal to the returned quantile is \( p \).

Note: GAMINV is the inverse of the PROBGAM function.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>q1=gaminv(0.5,9);</td>
<td>8.6689511844</td>
</tr>
<tr>
<td>q2=gaminv(0.1,2.1);</td>
<td>0.5841932369</td>
</tr>
</tbody>
</table>

GAMMA Function

Returns the value of the Gamma function
Category: Mathematical
Syntax
GAMMA(argument)

Arguments

argument is numeric.
Restriction: Nonpositive integers are invalid.

Details
The GAMMA function returns the integral, which is given by

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt.$$ For positive integers, \(\Gamma(x)\) is \((x - 1)!\). This function is commonly denoted by \(\Gamma(x)\).

Example

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=gamma(6);</td>
<td>120</td>
</tr>
</tbody>
</table>

GEOMEAN Function

Returns the geometric mean
Category: Descriptive Statistics

Syntax
GEOMEAN(argument<,argument,..>)

Arguments

argument is a non-negative numeric constant, variable, or expression.
Tip: The argument list can consist of a variable list, which is preceded by OF.

Comparisons

The MEAN function returns the arithmetic mean (average), and the HARMean function returns the harmonic mean, whereas the GEOMEAN function returns the geometric mean of the non-missing values. Unlike GEOMEANZ, GEOMEAN fuzzes the values of the arguments that are approximately zero.

Details

If any argument is negative, then the result is a missing value. A message appears in the log that the negative argument is invalid, and _ERROR_ is set to 1. If any argument is zero, then the geometric mean is zero. If all the arguments are missing values, then the result is a missing value. Otherwise, the result is the geometric mean of the non-missing values.

Let $n$ be the number of arguments with non-missing values, and let $x_1, x_2, \ldots, x_n$ be the values of those arguments. The geometric mean is the $n^{th}$ root of the product of the values:

$$\sqrt[n]{x_1 \cdot x_2 \cdot \ldots \cdot x_n}$$

Equivalently, the geometric mean is

$$\exp\left(\frac{\log(x_1) + \log(x_2) + \ldots + \log(x_n)}{n}\right)$$

Floating-point arithmetic often produces tiny numerical errors. Some computations that result in zero when exact arithmetic is used might result in a tiny non-zero value when floating-point arithmetic is used. Therefore, GEOMEAN fuzzes the values of arguments that are approximately zero. When the value of one argument is extremely small relative to the largest argument, then the former argument is treated as zero. If you do not want SAS to fuzz the extremely small values, then use the GEOMEANZ function.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1=geomean(1,2,2,4);</td>
<td>2</td>
</tr>
<tr>
<td>x2=geomean(.,2,4,8);</td>
<td>4</td>
</tr>
<tr>
<td>x3=geomean(of x1-x2);</td>
<td>2.8284271247</td>
</tr>
</tbody>
</table>
See Also

Function:
   “GEOMEANZ Function” on page 568
   “HARMEAN Function” on page 574
   “HARMEANZ Function” on page 575
   “MEAN Function” on page 644

GEOMEANZ Function

Returns the geometric mean, using zero fuzzing
Category: Descriptive Statistics

Syntax

GEOMEANZ(argument<,argument,…>)

Arguments

argument
   is a non-negative numeric constant, variable, or expression.
   Tip: The argument list can consist of a variable list, which is preceded by OF.

Comparisons

The MEAN function returns the arithmetic mean (average), and the HARMEAN function returns the harmonic mean, whereas the GEOMEANZ function returns the geometric mean of the non-missing values. Unlike GEOMEAN, GEOMEANZ does not fuzz the values of the arguments that are approximately zero.

Details

If any argument is negative, then the result is a missing value. A message appears in the log that the negative argument is invalid, and _ERROR_ is set to 1. If any argument is zero, then the geometric mean is zero. If all the arguments are missing values, then the result is a missing value. Otherwise, the result is the geometric mean of the non-missing values.

Let \( n \) be the number of arguments with non-missing values, and let \( x_1, x_2, \ldots, x_n \) be the values of those arguments. The geometric mean is the \( n^{th} \) root of the product of the values:

\[
\sqrt[n]{x_1 \times x_2 \times \ldots \times x_n}
\]
Equivalently, the geometric mean is

\[
\exp\left(\frac{\log(x_1) + \log(x_2) + \ldots + \log(x_n)}{n}\right)
\]

### Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1=geomeanz(1,2,2,4);</td>
<td>2</td>
</tr>
<tr>
<td>x2=geomeanz(.,2,4,8);</td>
<td>4</td>
</tr>
<tr>
<td>x3=geomeanz(of x1-x2);</td>
<td>2.8284271247</td>
</tr>
</tbody>
</table>

### See Also

Function:

“GEOMEAN Function” on page 566
“HARMEAN Function” on page 574
“HARMEANZ Function” on page 575
“MEAN Function” on page 644

### GETOPTION Function

Returns the value of a SAS system or graphics option

**Category:** Special

**Syntax**

\[ \text{GETOPTION}(\text{option-name}<,\text{reporting-options}<,...>) \]

**Arguments**

**option-name**

is the name of the system option.

**Tip:** Do not put an equals sign after the name. For example, write PAGESIZE= as PAGESIZE.

**Note:** SAS options that are passwords, such as EMAILPW and METAPASS, return the value x x x x x x x x, and not the actual password. △
reporting-options
specifies the reporting options. You can separate the options with blanks, or you can specify each reporting option as a separate argument to the GETOPTION function. The reporting options are

IN reports graphic units of measure in inches.
CM reports graphic units of measure in centimeters.
KEYWORD returns option values in a KEYWORD= format that would be suitable for direct use in the SAS OPTIONS or GOPTIONS global statements.

Examples

Example 1: Using GETOPTION to Change the YEARCUTOFF Option This example saves the initial value of the YEARCUTOFF option and then resets the value to 1920. The DATA step that follows verifies the option setting and performs date processing. When the DATA step ends, the YEARCUTOFF option is set to its original value.

```sas
%let cutoff=%sysfunc(getoption
  (yearcutoff,keyword));
options yearcutoff=1920;
data ages;
  if getoption('yearcutoff') = '1920' then
    do;
      ...more statements...
    end;
  else put 'Set Option YEARCUTOFF to 1920';
run;

options &cutoff;
```

Example 2: Using GETOPTION to Obtain Different Reporting Options This example defines a macro to illustrate the use of the GETOPTION function to obtain the value of system and graphics options by using different reporting options.

```sas
%macro showopts;
  %put PAGESIZE= %sysfunc(getoption(PAGESIZE));
  %put PS= %sysfunc(getoption(PS));
  %put LS= %sysfunc(getoption(LS));
  %put PS(keyword form)= %sysfunc(getoption(PS,keyword));
  %put LS(keyword form)= %sysfunc(getoption(LS,keyword));
  %put FORMCHAR= %sysfunc(getoption(FORMCHAR));
  %put HSIZE= %sysfunc(getoption(HSIZE));
  %put VSIZE= %sysfunc(getoption(VSIZE));
  %put HSIZE(in/keyword form)= %sysfunc(getoption(HSIZE,in,keyword));
  %put HSIZE(cm/keyword form)= %sysfunc(getoption(HSIZE,cm,keyword));
%mend showopts;
```
GETVARC Function

Returns the value of a SAS data set character variable

Category: SAS File I/O

Syntax

GETVARC(data-set-id, var-num)

Arguments

data-set-id
specifies the data set identifier that the OPEN function returns.

var-num
is the number of the variable in the Data Set Data Vector (DDV).

Tip: You can obtain this value by using the VARNUM function.

Tip: This value is listed next to the variable when you use the CONTENTS procedure.
Details

Use VARNUM to obtain the number of a variable in a SAS data set. VARNUM can be nested or it can be assigned to a variable that can then be passed as the second argument, as shown in the following examples. GETVARC reads the value of a character variable from the current observation in the Data Set Data Vector (DDV) into a macro or DATA step variable.

Examples

☐ This example opens the SASUSER.HOUSES data set and gets the entire tenth observation. The data set identifier value for the open data set is stored in the macro variable MYDATAID. This example nests VARNUM to return the position of the variable in the DDV, and reads in the value of the character variable STYLE.

```sas
%let mydataid=%sysfunc(open
  (sasuser.houses,i));
%let rc=%sysfunc(fetchobs(&mydataid,10));
%let style=%sysfunc(getvarc(&mydataid,
  %sysfunc(varnum
      (&mydataid,STYLE))));
%let rc=%sysfunc(close(&mydataid));
```

☐ This example assigns VARNUM to a variable that can then be passed as the second argument. This example fetches data from observation 10.

```sas
%let namenum=%sysfunc(varnum(&mydataid,NAME));
%let rc=%sysfunc(fetchobs(&mydataid,10));
%let user=%sysfunc(getvarc
      (&mydataid,&namenum));
```

See Also

Functions:

“FETCH Function” on page 522
“FETCHOBS Function” on page 523
“GETVARN Function” on page 572
“VARNUM Function” on page 878

GETVARN Function

Returns the value of a SAS data set numeric variable

Category: SAS File I/O

Syntax

```sas
GETVARN(data-set-id,var-num)
```
Arguments

data-set-id
specifies the data set identifier that the OPEN function returns.

var-num
is the number of the variable in the Data Set Data Vector (DDV).

Tip: You can obtain this value by using the VARNUM function.

Tip: This value is listed next to the variable when you use the CONTENTS procedure.

Details

Use VARNUM to obtain the number of a variable in a SAS data set. You can nest VARNUM or you can assign it to a variable that can then be passed as the second argument, as shown in the "Examples" section. GETVARN reads the value of a numeric variable from the current observation in the Data Set Data Vector (DDV) into a macro variable or DATA step variable.

Examples

□ This example obtains the entire tenth observation from a SAS data set. The data set must have been previously opened using OPEN. The data set identifier value for the open data set is stored in the variable MYDATAID. This example nests VARNUM, and reads in the value of the numeric variable PRICE from the tenth observation of an open SAS data set.

```sas
%let rc=%sysfunc(fetchobs(&mydataid,10));
%let price=%sysfunc(getvarn(&mydataid,
%sysfunc(varnum
(&mydataid,price))));
```

□ This example assigns VARNUM to a variable that can then be passed as the second argument. This example fetches data from observation 10.

```sas
%let pricenum=%sysfunc(varnum
(&mydataid,price));
%let rc=%sysfunc(fetchobs(&mydataid,10));
%let price=%sysfunc(getvarn
(&mydataid,&pricenum));
```

See Also

Functions:

“FETCH Function” on page 522
“FETCHOBS Function” on page 523
“GETVARC Function” on page 571
“VARNUM Function” on page 878
HARMEAN Function

Returns the harmonic mean

Category: Descriptive Statistics

Syntax

HARMEAN(argument<,argument,…>)

Arguments

argument

is a non-negative numeric constant, variable, or expression.

Tip: The argument list can consist of a variable list, which is preceded by OF.

Comparisons

The MEAN function returns the arithmetic mean (average), and the GEOMEAN function returns the geometric mean, whereas the HARMEAN function returns the harmonic mean of the non-missing values. Unlike HARMEANZ, HARMEAN fuzzes the values of the arguments that are approximately zero.

Details

If any argument is negative, then the result is a missing value. A message appears in the log that the negative argument is invalid, and _ERROR_ is set to 1. If all the arguments are missing values, then the result is a missing value. Otherwise, the result is the harmonic mean of the non-missing values.

If any argument is zero, then the harmonic mean is zero. Otherwise, the harmonic mean is the reciprocal of the arithmetic mean of the reciprocals of the values.

Let \( n \) be the number of arguments with non-missing values, and let \( x_1, x_2, \ldots, x_n \) be the values of those arguments. The harmonic mean is

\[
\frac{1}{x_1} + \frac{1}{x_2} + \ldots + \frac{1}{x_n}
\]

Floating-point arithmetic often produces tiny numerical errors. Some computations that result in zero when exact arithmetic is used might result in a tiny non-zero value when floating-point arithmetic is used. Therefore, HARMEAN fuzzes the values of arguments that are approximately zero. When the value of one argument is extremely small relative to the largest argument, then the former argument is treated as zero. If you do not want SAS to fuzz the extremely small values, then use the HARMEANZ function.
### Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1=harmean(1,2,4,4);</td>
<td>2</td>
</tr>
<tr>
<td>x2=harmean(.,4,12,24);</td>
<td>8</td>
</tr>
<tr>
<td>x3=harmean(of x1-x2);</td>
<td>3.2</td>
</tr>
</tbody>
</table>

### See Also

Function:

- “GEOMEAN Function” on page 566
- “GEOMEANZ Function” on page 568
- “HARMEANZ Function” on page 575
- “MEAN Function” on page 644

### HARMEANZ Function

Returns the harmonic mean, using zero fuzzing

**Category:** Descriptive Statistics

**Syntax**

```
HARMEANZ(argument<,argument,...>)
```

**Arguments**

`argument`

is a non-negative numeric constant, variable, or expression.

**Tip:** The argument list can consist of a variable list, which is preceded by **OF**.

**Comparisons**

The **MEAN** function returns the arithmetic mean (average), and the **GEOMEAN** function returns the geometric mean, whereas the **HARMEANZ** function returns the harmonic mean of the non-missing values. Unlike **HARMEAN**, **HARMEANZ** does not fuzz the values of the arguments that are approximately zero.

**Details**

If any argument is negative, then the result is a missing value. A message appears in the log that the negative argument is invalid, and **_ERROR_** is set to 1. If all the
arguments are missing values, then the result is a missing value. Otherwise, the result is the harmonic mean of the non-missing values.

If any argument is zero, then the harmonic mean is zero. Otherwise, the harmonic mean is the reciprocal of the arithmetic mean of the reciprocals of the values.

Let \( n \) be the number of arguments with non-missing values, and let \( x_1, x_2, \ldots, x_n \) be the values of those arguments. The harmonic mean is

\[
\frac{1}{x_1} + \frac{1}{x_2} + \ldots + \frac{1}{x_n}
\]

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1=harmeanz(1,2,4,4);</td>
<td>2</td>
</tr>
<tr>
<td>x2=harmeanz(.,4,12,24);</td>
<td>8</td>
</tr>
<tr>
<td>x3=harmeanz(of x1-x2);</td>
<td>3.2</td>
</tr>
</tbody>
</table>

See Also

Function:
- “GEOMEAN Function” on page 566
- “GEOMEANZ Function” on page 568
- “HARMEAN Function” on page 574
- “MEAN Function” on page 644

HBOUND Function

Returns the upper bound of an array

Category: Array

Syntax

HBOUND\(<n>\)(array-name)

HBOUND(array-name,bound-n)

Arguments

\( n \)
specifies the dimension for which you want to know the upper bound. If no \( n \) value is specified, the HBOUND function returns the upper bound of the first dimension of the array.

\textit{array-name}

specifies the name of an array defined previously in the same DATA step.

\textit{bound-n}

specifies the dimension for which you want to know the upper bound. Use \textit{bound-n} only if \( n \) is not specified.

**Details**

The HBOUND function returns the upper bound of a one-dimensional array or the upper bound of a specified dimension of a multidimensional array. Use HBOUND in array processing to avoid changing the upper bound of an iterative DO group each time you change the bounds of the array. HBOUND and LBOUND can be used together to return the values of the upper and lower bounds of an array dimension.

**Comparisons**

- HBOUND returns the literal value of the upper bound of an array dimension.
- DIM always returns a total count of the number of elements in an array dimension.

\textit{Note:} This distinction is important when the lower bound of an array dimension has a value other than 1 and the upper bound has a value other than the total number of elements in the array dimension.

**Examples**

**Example 1: One-dimensional Array**

In this example, HBOUND returns the upper bound of the dimension, a value of 5. Therefore, SAS repeats the statements in the DO loop five times.

\begin{verbatim}
array big{5} weight sex height state city;
do i=1 to hbound(big5);
  more SAS statements;
end;
\end{verbatim}

**Example 2: Multidimensional Array**

This example shows two ways of specifying the HBOUND function for multidimensional arrays. Both methods return the same value for HBOUND, as shown in the table that follows the SAS code example.

\begin{verbatim}
array mult{2:6,4:13,2} multi-multi100;
\end{verbatim}

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Alternative Syntax</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>HBOUND(MULT)</td>
<td>HBOUND(MULT,1)</td>
<td>6</td>
</tr>
<tr>
<td>HBOUND2(MULT)</td>
<td>HBOUND(MULT,2)</td>
<td>13</td>
</tr>
<tr>
<td>HBOUND3(MULT)</td>
<td>HBOUND(MULT,3)</td>
<td>2</td>
</tr>
</tbody>
</table>
See Also

Functions:

“DIM Function” on page 498
“LBOUND Function” on page 624

Statements:

“ARRAY Statement” on page 1102
“Array Reference Statement” on page 1107
“Array Processing” in SAS Language Reference: Concepts

HMS Function

Returns a SAS time value from hour, minute, and second values

Category: Date and Time

Syntax

HMS(hour, minute, second)

Arguments

hour
is numeric.

minute
is numeric.

second
is numeric.

Details

The HMS function returns a positive numeric value that represents a SAS time value.

Examples

The following SAS statements produce these results:

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>hrid=hms(12,45,10); put hrid / hrid time.;</td>
<td>45910 2:45:10</td>
</tr>
</tbody>
</table>
See Also

Functions:

“DHMS Function” on page 495
“HOUR Function” on page 579
“MINUTE Function” on page 646
“SECOND Function” on page 820

HOUR Function

Returns the hour from a SAS time or datetime value

Category: Date and Time

Syntax

HOUR(<time | datetime>)

Arguments

*time*

specifies a SAS expression that represents a SAS time value.

*datetime*

specifies a SAS expression that represents a SAS datetime value.

Details

The HOUR function returns a numeric value that represents the hour from a SAS time or datetime value. Numeric values can range from 0 through 23. HOUR always returns a positive number.

Examples

The following SAS statements produce these results:

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>now='1:30't;</td>
<td></td>
</tr>
<tr>
<td>h=hour(now);</td>
<td></td>
</tr>
<tr>
<td>put h;</td>
<td>1</td>
</tr>
</tbody>
</table>
See Also

Functions:
“MINUTE Function” on page 646
“SECOND Function” on page 820

HTMLDECODE Function

Decodes a string containing HTML numeric character references or HTML character entity references and returns the decoded string

Category: Web Tools

Syntax

HTMLDECODE(expression)

Arguments

expression
specifies any character expression.

Details

The HTMLDECODE function recognizes the following character entity references:

<table>
<thead>
<tr>
<th>Character entity reference</th>
<th>decoded character</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>&amp;</td>
</tr>
<tr>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>&quot;</td>
<td>&quot;</td>
</tr>
<tr>
<td>'*</td>
<td>’</td>
</tr>
</tbody>
</table>

* (SAS 9 only)

Unrecognized entities (&<name>;) are left unmodified in the output string. The HTMLDECODE function recognizes numeric entity references that are of the form

&#nnn; where nnn specifies a decimal number that contains one or more digits.

&amp;#Xnnn; where nnn specifies a hexadecimal number that contains one or more digits.
Operating Environment Information: Numeric character references that cannot be represented in the current SAS session encoding will not be decoded. The reference will be copied unchanged to the output string.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1=htmldecode('not a &lt;tag&gt;');</td>
<td>not a &lt;tag&gt;</td>
</tr>
<tr>
<td>x2=htmldecode('&amp;');</td>
<td>'&amp;xxx'</td>
</tr>
<tr>
<td>x3=htmldecode ('ABC');</td>
<td>'ABC'</td>
</tr>
</tbody>
</table>

See Also

Function:
“HTMLENCODE Function” on page 581

---

HTMLENCODE Function

Encodes characters using HTML character entity references and returns the encoded string

Category: Web Tools

Syntax

HTMLENCODE(expression, <options>)

Arguments

expression

specifies any character expression. By default, any greater-than (>), less-than (<), and ampersand (&) characters are encoded as &gt;, &lt;, and &amp;, respectively. In SAS 9 only, this behavior can be modified with the options argument.

Note: The encoded string can be longer than the original string. You should take the additional length into consideration when you define your output variable. If the encoded string exceeds the maximum length that is defined, the output string might be truncated.

options

specifies the type of characters to encode. The value can be any string expression. If you use more than one option, separate the options by spaces. See “Details” on page 582 for a list of valid options.
Note: This argument is available in SAS 9 only.

Details

<table>
<thead>
<tr>
<th>Option</th>
<th>Character</th>
<th>Character Entity Reference</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>amp</td>
<td>&amp;</td>
<td>&amp;</td>
<td>The HTMLENCODE function encodes these characters by default. If you need to encode these characters only, then you do not need to specify the options argument. However, if you specify any value for the options argument, then the defaults are overridden, and you must explicitly specify the options for all of the characters you want to encode.</td>
</tr>
<tr>
<td>gt</td>
<td>&gt;</td>
<td>&gt;</td>
<td>Use this option to encode the apostrophe (') character in text that is used in an HTML or XML tag attribute.</td>
</tr>
<tr>
<td>lt</td>
<td>&lt;</td>
<td>&lt;</td>
<td>Use this option to encode the double quotation mark (&quot;) character in text that is used in an HTML or XML tag attribute.</td>
</tr>
<tr>
<td>apos</td>
<td>'</td>
<td>'</td>
<td>Use this option to encode the 7-bit any character that is not represented in 7-bit ASCII encoding</td>
</tr>
<tr>
<td>quot</td>
<td>&quot;</td>
<td>&quot;</td>
<td>nnn is a one or more digit hexadecimal number. Encode these characters to create HTML or XML that is easily transferred through communication paths that might only support 7-bit ASCII encodings (for example, ftp or e-mail).</td>
</tr>
</tbody>
</table>

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>htmlencode(&quot;John’s test &lt;tag&gt;&quot;)</td>
<td>John’s test &lt;tag&gt;</td>
</tr>
<tr>
<td>htmlencode(&quot;John’s test &lt;tag&gt;&quot;, ‘apos’)</td>
<td>John's test &lt;tag&gt;</td>
</tr>
<tr>
<td>htmlencode(‘John “Jon” Smith &lt;tag&gt;’, ‘quot’)</td>
<td>John &quot;Jon&quot; Smith &lt;tag&gt;</td>
</tr>
<tr>
<td>htmlencode(‘”A&amp;B&amp;C”’, ‘amp lt gt apos’)</td>
<td>'A&amp;B&amp;C&amp;apos;</td>
</tr>
<tr>
<td>htmlencode(’80’x, ‘7bit’) (’80’x is the euro symbol in Western European locales.)</td>
<td>€</td>
</tr>
</tbody>
</table>

See Also

Function:
IBESSEL Function

Returns the value of the modified bessel function

Category: Mathematical

Syntax

IBESSEL(nu,x,kode)

Arguments

nu
is numeric.
Range: \( \nu \geq 0 \)

x
is numeric.
Range: \( x \geq 0 \)

kode
is a nonnegative integer.

Details

The IBESSEL function returns the value of the modified bessel function of order \( \nu \) evaluated at \( x \) (Abramowitz, Stegun 1964; Amos, Daniel, Weston 1977). When \( kode \) equals 0, the bessel function is returned. Otherwise, the value of the following function is returned:

\[
e^{-x} I_{nm}(x)
\]

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=ibessel(2,2,0);</td>
<td>0.6889484477</td>
</tr>
<tr>
<td>x=ibessel(2,2,1);</td>
<td>0.0932390333</td>
</tr>
</tbody>
</table>

IFC Function

Returns a character value of an expression based on whether the expression is true, false, or missing
Category: Character

Syntax

IFC(logical-expression, value-returned-when-true, value-returned-when-false
    <,value-returned-when-missing>)

Arguments

logical-expression
    specifies a numeric expression.

value-returned-when-true
    specifies a character expression that is returned when the value of logical-expression
    is true.

value-returned-when-false
    specifies a character expression that is returned when the value of logical-expression
    is false.

value-returned-when-missing
    specifies a character expression that is returned when the value of logical-expression
    is missing.

Details

The IFC function uses conditional logic that enables you to select among several
    different values based on the value of a logical expression.

    IFC evaluates the first argument, logical-expression. If logical-expression is true (that
    is, not zero and not missing), then IFC returns the value in the second argument. If
    logical-expression is a missing value, and you have a fourth argument, then IFC returns
    the value in the fourth argument. If logical-expression is false, IFC returns the value in
    the third argument.

    The IFC function is useful in DATA step expressions, and even more useful in
    WHERE clauses and other expressions where it is not convenient or possible to use an
    IF/THEN/ELSE construct.

Comparisons

The IFC function is similar to the IFN function except that IFC returns a character
    value while IFN returns a numeric value.

Examples

In the following example, IFC evaluates the expression grade>80 to implement the
    logic that determines the performance of several members on a team. The results are
    written to the SAS log.

```sas
   data _null_;
      input name $ grade;
      performance = ifc(grade>80, 'Pass ', 'Needs Improvement');
      put name= performance=;"
This example uses an IF/THEN/ELSE construct to generate the same output that is generated by the IFC function. The results are written to the SAS log.

```sas
data _null_;
  input name $ grade;
  if grade>80 then performance='Pass';
    else performance='Needs Improvement';
  put name= performance=;
datalines;
John 74
Sam 89
Kati 100
Maria 92
;
run;
```

### See Also

Functions:

“IFN Function” on page 585

### IFN Function

Returns a numeric value of an expression based on whether the expression is true, false, or missing.
Category: Character

Syntax

\[ \text{IFN}(\text{logical-expression}, \text{value-returned-when-true}, \text{value-returned-when-false}, \text{value-returned-when-missing}) \]

Arguments

- **logical-expression**
  - specifies a numeric expression.
- **value-returned-when-true**
  - specifies a numeric expression that is returned when the value of logical-expression is true.
- **value-returned-when-false**
  - specifies a numeric expression that is returned when the value of logical-expression is false.
- **value-returned-when-missing**
  - specifies a numeric expression that is returned when the value of logical-expression is missing.

Details

The IFN function uses conditional logic that enables you to select among several different values based on the value of a logical expression.

IFN evaluates the first argument, then logical-expression. If logical-expression is true (that is, not zero and not missing), then IFN returns the value in the second argument. If logical-expression is a missing value, and you have a fourth argument, then IFN returns the value in the fourth argument. If logical-expression is false, IFN returns the value in the third argument.

The IFN function, an IF/THEN/ELSE construct, or a WHERE statement can produce the same results (see “Examples” on page 586). However, the IFN function is useful in DATA step expressions when it is not convenient or possible to use an IF/THEN/ELSE construct or a WHERE statement.

Comparisons

The IFN function is similar to the IFC function, except that IFN returns a numeric value whereas IFC returns a character value.

Examples

**Example 1: Calculating Sales Commission**

The following examples show how to calculate sales commission using the IFN function, an IF/THEN/ELSE construct, and a WHERE statement. In each of the examples, the commission that is calculated is the same.

Calculating Commission Using the IFN Function

In the following example, IFN evaluates the expression \text{TotalSales} > 10000. If total sales exceeds $10,000, then the
sales commission is 5% of the total sales. If total sales is less than $10,000, then the sales commission is 2% of the total sales.

```sas
data _null_
  input TotalSales;
  commission=ifn(TotalSales > 10000, TotalSales*.05, TotalSales*.02);
  put commission=;
datalines;
25000
10000
500
10300
;
run;
```

SAS writes the following output to the log:

```
commission=1250
commission=200
commission=10
commission=515
```

Calculating Commission Using an IF/THEN/ELSE Construct  
In the following example, an IF/THEN/ELSE construct evaluates the expression `TotalSales > 10000`. If total sales exceeds $10,000, then the sales commission is 5% of the total sales. If total sales is less than $10,000, then the sales commission is 2% of the total sales.

```sas
data _null_
  input TotalSales;
  if TotalSales > 10000 then commission = .05 * TotalSales;
  else commission = .02 * TotalSales;
  put commission=;
datalines;
25000
10000
500
10300
;
run;
```

SAS writes the following output to the log:

```
commission=1250
commission=200
commission=10
commission=515
```

Calculating Commission Using a WHERE Statement  
In the following example, a WHERE statement evaluates the expression `TotalSales > 10000`. If total sales exceeds $10,000, then the sales commission is 5% of the total sales. If total sales is less than $10,000, then the sales commission is 2% of the total sales. The output shows only those salespeople whose total sales exceed $10,000.
options pageno=1 nodate ls=80 ps=64;

data sales;
   input SalesPerson $ TotalSales;
   datalines;
Michaels 25000
Janowski 10000
Chen 500
Gupta 10300
;

data commission;
   set sales;
   where TotalSales > 10000;
   commission = TotalSales * .05;
run;

proc print data=commission;
   title 'Commission for Total Sales > 1000';
run;

Output 4.27  Output from a WHERE Statement

<table>
<thead>
<tr>
<th>Obs</th>
<th>Person</th>
<th>Sales</th>
<th>commission</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Michaels</td>
<td>25000</td>
<td>1250</td>
</tr>
<tr>
<td>2</td>
<td>Gupta</td>
<td>10300</td>
<td>515</td>
</tr>
</tbody>
</table>

See Also

Functions:

“IFC Function” on page 583

INDEX Function

Searches a character expression for a string of characters

Category: Character

Syntax

INDEX(source,excerpt)
Arguments

source
  specifies the character expression to search.

excerpt
  specifies the string of characters to search for in the character expression.

  Tip:  Enclose a literal string of characters in quotation marks.

Details

The INDEX function searches source, from left to right, for the first occurrence of the string specified in excerpt, and returns the position in source of the string’s first character. If the string is not found in source, INDEX returns a value of 0. If there are multiple occurrences of the string, INDEX returns only the position of the first occurrence.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a='ABC.DEF (X=Y)'</td>
<td></td>
</tr>
<tr>
<td>b='X=Y';</td>
<td></td>
</tr>
<tr>
<td>x=index(a,b);</td>
<td></td>
</tr>
<tr>
<td>put x;</td>
<td>10</td>
</tr>
</tbody>
</table>

See Also

Functions:

  “INDEXC Function” on page 589
  “INDEXW Function” on page 590

INDEXC Function

Searches a character expression for specific characters

Category: Character

Syntax

INDEXC(source,excerpt-1,<... excerpt-n>)
Arguments

*source*

specifies the character expression to search.

*excerpt*

specifies the characters to search for in the character expression.

**Tip:** If you specify more than one excerpt, separate them with a comma.

Details

The INDEXC function searches *source*, from left to right, for the first occurrence of any character present in the excerpts and returns the position in *source* of that character. If none of the characters in *excerpt-1* through *excerpt-n* in *source* are found, INDEXC returns a value of 0.

Comparisons

The INDEXC function searches for the first occurrence of any individual character that is present within the character string, whereas the INDEX function searches for the first occurrence of the character string as a pattern.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a='ABC.DEP (X2=Y1)';</td>
<td></td>
</tr>
<tr>
<td>x=indexc(a,'0123',';()=.');</td>
<td></td>
</tr>
<tr>
<td>put x;</td>
<td>4</td>
</tr>
<tr>
<td>b='have a good day';</td>
<td></td>
</tr>
<tr>
<td>x=indexc(b,'pleasant','very');</td>
<td></td>
</tr>
<tr>
<td>put x;</td>
<td>2</td>
</tr>
</tbody>
</table>

See Also

Functions:

“INDEX Function” on page 588

“INDEXW Function” on page 590

<table>
<thead>
<tr>
<th>INDEXW Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Searches a character expression for a specified string as a word</td>
</tr>
<tr>
<td>Category: Character</td>
</tr>
</tbody>
</table>
Syntax

INDEXW(source, excerpt<delimiter>)

Arguments

source
specifies the character expression to search.

delimiter
specifies a character expression that you want INDEXW to use as a word separator in the character string. The default delimiter is the blank character.

Details

The INDEXW function searches source, from left to right, for the first occurrence of excerpt and returns the position in source of the substring’s first character. If the substring is not found in source, then INDEXW returns a value of 0. If there are multiple occurrences of the string, then INDEXW returns only the position of the first occurrence.

The substring pattern must begin and end on a word boundary. For INDEXW, word boundaries are delimiters, the beginning of source, and the end of source.

INDEXW has the following behavior when the second argument contains blank spaces or has a length of 0:

- If both source and excerpt contain only blank spaces or have a length of 0, then INDEXW returns a value of 1.
- If excerpt contains only blank spaces or has a length of 0, and source contains character or numeric data, then INDEXW returns a value of 0.

Comparisons

The INDEXW function searches for strings that are words, whereas the INDEX function searches for patterns as separate words or as parts of other words. INDEXC searches for any characters that are present in the excerpts.

Examples

The following SAS statements give these results.

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>s='asdf adog dog'; p='dog'; x=indexw(s,p); put x;</td>
<td>11</td>
</tr>
<tr>
<td>s='abcdef x=y'; p='def'; x=indexw(s,p); put x;</td>
<td>0</td>
</tr>
</tbody>
</table>
### INPUT Function

**INPUT Function**

Returns the value produced when a SAS expression that uses a specified informat expression is read.

**Category:** Special

**Syntax**

\[
\text{INPUT}(source, <? | ??>) \text{informat.})
\]

**Arguments**

\*source\* contains the SAS character expression to which you want to apply a specific informat.

\*? or ??\* The optional question mark (?) and double question mark (??) format modifiers suppress the printing of both the error messages and the input lines when invalid.
data values are read. The ? modifier suppresses the invalid data message. The ?? modifier also suppresses the invalid data message and, in addition, prevents the automatic variable _ERROR_ from being set to 1 when invalid data are read.

**informat** is the SAS informat that you want to apply to the source.

**Details**

If the INPUT function returns a value to a variable that has not yet been assigned a length, by default the variable length is determined by the width of the informat. The INPUT function enables you to read the value of source by using a specified informat. The informat determines whether the result is numeric or character. Use INPUT to convert character values to numeric values.

**Comparisons**

The INPUT function returns the value produced when a SAS expression is read using a specified informat. You must use an assignment statement to store that value in a variable. The INPUT statement uses an informat to read a data value and then optionally stores that value in a variable.

**Examples**

**Example 1: Converting Character Values to Numeric Values**

This example uses the INPUT function to convert a character value to a numeric value and store it in another variable. The COMMA9. informat reads the value of the SALE variable, stripping the commas. The resulting value, 2115353, is stored in FMTSALE.

```sas
data testin;
  input sale $9.;
  fmtsale=input(sale, comma9.);
  datalines;
  2,115,353
;
```

**Example 2: Using PUT and INPUT Functions**

In this example, PUT returns a numeric value as a character string. The value 122591 is assigned to the CHARDATE variable. INPUT returns the value of the character string as a SAS date value using a SAS date informat. The value 11681 is stored in the SASDATE variable.

```sas
numdate=122591;
chardate=put(numdate,z6.);
sasdate=input(chardate, mmddyy6.);
```

**Example 3: Suppressing Error Messages**

In this example, the question mark (?) format modifier tells SAS not to print the invalid data error message if it finds data errors. The automatic variable _ERROR_ is set to 1 and input data lines are written to the SAS log.

```sas
y=input(x, ? 3.1);
```

Because the double question mark (??) format modifier suppresses printing of error messages and input lines and prevents the automatic variable _ERROR_ from being set to 1 when invalid data are read, the following two examples produce the same result:

- `y=input(x, ?? 2.);
- `y=input(x, ? 2.); _error_=0;`
INPUTC Function

Enables you to specify a character informat at run time

Category: Special

Syntax

INPUTC(source, informat.<,w>)

Arguments

source
is the SAS expression to which you want to apply the informat.

informat.
is an expression that contains the character informat you want to apply to source.

w
specifies a width to apply to the informat.

Interaction: If you specify a width here, it overrides any width specification in the informat.

Details

If the INPUTC function returns a value to a variable that has not yet been assigned a length, by default the variable length is determined by the length of the first argument.

Comparisons

The INPUTN function enables you to specify a numeric informat at run time.

Examples

Example 1: Specifying Character Informats
The PROC FORMAT step in this example creates a format, TYPEFMT., that formats the variable values 1, 2, and 3 with the
The name of one of the three informats that this step also creates. The informats store responses of "positive," "negative," and "neutral" as different words, depending on the type of question. After PROC FORMAT creates the format and informats, the DATA step creates a SAS data set from raw data consisting of a number identifying the type of question and a response. After reading a record, the DATA step uses the value of TYPE to create a variable, RESPINF, that contains the value of the appropriate informat for the current type of question. The DATA step also creates another variable, WORD, whose value is the appropriate word for a response. The INPUTC function assigns the value of WORD based on the type of question and the appropriate informat.

```sas
proc format;
   value typefmt 1='$groupx'
          2='$groupy'
          3='$groupz';
   invalue $groupx 'positive'='agree'
                  'negative'='disagree'
                  'neutral'='notsure';
   invalue $groupy 'positive'='accept'
                  'negative'='reject'
                  'neutral'='possible';
   invalue $groupz 'positive'='pass'
                  'negative'='fail'
                  'neutral'='retest';
run;

data answers;
   input type response $;
   respinformat = put(type, typefmt.);
   word = inputc(response, respinformat);
   datalines;
1 positive
1 negative
1 neutral
2 positive
2 negative
2 neutral
3 positive
3 negative
3 neutral
;
```

The value of WORD for the first observation is agree. The value of WORD for the last observation is retest.

### See Also

**Functions:**

- "INPUT Function" on page 592
- "INPUTN Function" on page 596
- "PUT Function" on page 754
- "PUTC Function" on page 756
- "PUTN Function" on page 758
INPUTN Function

Enables you to specify a numeric informat at run time

Category: Special

Syntax

\[ \text{INPUTN}(\text{source}, \text{informat}.<,\text{w}<,\text{d}>) \]

Arguments

\textit{source}

is the SAS expression to which you want to apply the informat.

\textit{informat.}

is an expression that contains the numeric informat you want to apply to source.

\textit{w}

specifies a width to apply to the informat.

Interaction: If you specify a width here, it overrides any width specification in the informat.

\textit{d}

specifies the number of decimal places to use.

Interaction: If you specify a number here, it overrides any decimal-place specification in the informat.

Comparisons

The INPUTC function enables you to specify a character informat at run time.

Examples

Example 1: Specifying Numeric Informats

The PROC FORMAT step in this example creates a format, READDATE., that formats the variable values 1 and 2 with the name of a SAS date informat. The DATA step creates a SAS data set from raw data originally from two different sources (indicated by the value of the variable SOURCE). Each source specified dates differently. After reading a record, the DATA step uses the value of SOURCE to create a variable, DATEINF, that contains the value of the appropriate informat for reading the date. The DATA step also creates a new variable, NEWDATE, whose value is a SAS date. The INPUTN function assigns the value of NEWDATE based on the source of the observation and the appropriate informat.

\begin{verbatim}
proc format;
   value readdate 1='date7.'
       2='mmddyy8.';
run;

options yearcutoff=1920;

data fixdates (drop=start dateinformat);
\end{verbatim}
**INT Function**

**Returns the integer value, fuzzed to avoid unexpected floating-point results**

**Category:** Truncation

---

**Syntax**

`INT(argument)`

**Arguments**

`argument`

is numeric.

**Details**

The `INT` function returns the integer portion of the argument (truncates the decimal portion). If the argument's value is within 1E-12 of an integer, the function results in that integer. If the value of `argument` is positive, the `INT` function has the same result as the `FLOOR` function. If the value of `argument` is negative, the `INT` function has the same result as the `CEIL` function.

**Comparisons**

Unlike the `INTZ` function, the `INT` function fuzzes the result. If the argument is within 1E-12 of an integer, the `INT` function fuzzes the result to be equal to that integer. The
INTCK function does not fuzz the result. Therefore, with the INTZ function you might get unexpected results.

**Examples**

The following SAS statements produce these results.

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>var1=2.1;</td>
<td></td>
</tr>
<tr>
<td>x=int(var1);</td>
<td></td>
</tr>
<tr>
<td>put x;</td>
<td>2</td>
</tr>
<tr>
<td>var2=-2.4;</td>
<td></td>
</tr>
<tr>
<td>y=int(var2);</td>
<td></td>
</tr>
<tr>
<td>put y;</td>
<td>-2</td>
</tr>
<tr>
<td>a=int(1+1.e-11);</td>
<td></td>
</tr>
<tr>
<td>put a;</td>
<td>1</td>
</tr>
<tr>
<td>b=int(-1.6);</td>
<td></td>
</tr>
<tr>
<td>put b;</td>
<td>-1</td>
</tr>
</tbody>
</table>

**See Also**

Functions:

“CEIL Function” on page 429
“FLOOR Function” on page 544
“INTZ Function” on page 608

---

**INTCK Function**

Returns the integer count of the number of interval boundaries between two dates, two times, or two datetime values

**Category:** Date and Time

**Syntax**

`INTCK(interval, from, to)`

**Arguments**

`interval`

specifies a character constant, a variable, or an expression that contains an interval name. `Interval` can appear in uppercase or lowercase. The possible values of `interval` are listed in the “Intervals Used with Date and Time Functions” table in SAS Language Reference: Concepts.
**Requirement:** The type of interval (date, datetime, or time) must match the type of value in `from`.

Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications. The general form of an interval name is as follows:

```
interval<n.s>
```

The three parts of the interval name are

- **interval**
  - specifies the name of the basic interval type. For example, `YEAR` specifies yearly intervals.
- **n**
  - specifies an optional multiplier that sets the interval equal to a multiple of the period of the basic interval type. For example, the interval `YEAR2` consists of two-year, or biennial, periods.
  - See: “Details” on page 599 for more information.
- **s**
  - specifies an optional shift index that shifts the interval to start at a specified subperiod starting point. For example, `YEAR.3` specifies yearly periods shifted to start on the first of March of each calendar year and to end in February of the following year.
  - Restriction: The shift index cannot be greater than the number of subperiods in the whole interval. For example, you could use `YEAR2.24`, but `YEAR2.25` would be an error because there is no twenty-fifth month in a two-year interval.
  - Restriction: If the default shift period is the same as the interval type, then only multiperiod intervals can be shifted with the optional shift index. For example, `MONTH` type intervals shift by `MONTH` subperiods by default; thus, monthly intervals cannot be shifted with the shift index. However, bimonthly intervals can be shifted with the shift index, because there are two `MONTH` intervals in each `MONTH2` interval. The interval name `MONTH2.2`, for example, specifies bimonthly periods starting on the first day of even-numbered months.
  - See: “Details” on page 599 for more information.

- **from**
  - specifies a SAS expression that represents the starting SAS date, time, or datetime value.

- **to**
  - specifies a SAS expression that represents the ending SAS date, time, or datetime value.

**Details**

The INTCK function counts the number of times the beginning of an interval is reached in moving from the `from` argument to the `to` argument. It does not count the number of complete intervals between those two values. For example, WEEK intervals are determined by the number of Sundays that occur between the `from` argument and the `to` argument, not by how many seven-day periods fall in between the `from` argument and the `to` argument.
Both the multiplier \( n \) and the shift index \( s \) are optional and default to 1. For example, \texttt{YEAR}, \texttt{YEAR1}, \texttt{YEAR.1}, and \texttt{YEAR1.1} are all equivalent ways of specifying ordinary calendar years.

Intervals that do not nest within years or days are aligned relative to the SAS date or datetime value 0. The arbitrary reference time of midnight on January 1, 1960, is used as the origin for non-shifted intervals, and shifted intervals are defined relative to that reference point.

For example, \texttt{MONTH13} defines the intervals January 1, 1960, February 1, 1961, March 1, 1962, and so on, and the intervals December 1, 1958, November 1, 1957, and so on, before the base date January 1, 1960.

As another example, the interval specification \texttt{WEEK6.13} defines six-week periods starting on second Fridays, and the convention of alignment relative to the period containing January 1, 1960, tells where to start counting to find out what dates correspond to the second Fridays of six-week intervals.

<table>
<thead>
<tr>
<th>Interval</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{DAY3}</td>
<td>Three-day intervals starting on Sunday</td>
</tr>
<tr>
<td>\texttt{WEEK.7}</td>
<td>Weekly with Saturday as the first day of the week</td>
</tr>
<tr>
<td>\texttt{WEEK6.13}</td>
<td>Six-week intervals starting on second Fridays</td>
</tr>
<tr>
<td>\texttt{WEEK2}</td>
<td>Biweekly intervals starting on first Sundays</td>
</tr>
<tr>
<td>\texttt{WEEK1.1}</td>
<td>Same as \texttt{WEEK}</td>
</tr>
<tr>
<td>\texttt{WEEK.2}</td>
<td>Weekly intervals starting on Mondays</td>
</tr>
<tr>
<td>\texttt{WEEK6.3}</td>
<td>Six-week intervals starting on first Tuesdays</td>
</tr>
<tr>
<td>\texttt{WEEK6.11}</td>
<td>Six-week intervals starting on second Wednesdays</td>
</tr>
<tr>
<td>\texttt{WEEK4}</td>
<td>Four-week intervals starting on first Sundays</td>
</tr>
<tr>
<td>\texttt{WEEKDAY1W}</td>
<td>Six-day week with Sunday as a weekend day</td>
</tr>
<tr>
<td>\texttt{WEEKDAY35W}</td>
<td>Five-day week with Tuesday and Thursday as weekend days (W indicates that day 3 and day 5 are weekend days)</td>
</tr>
<tr>
<td>\texttt{WEEKDAY17W}</td>
<td>Same as \texttt{WEEKDAY}</td>
</tr>
<tr>
<td>\texttt{WEEKDAY67W}</td>
<td>Five-day week with Friday and Saturday as weekend days</td>
</tr>
<tr>
<td>\texttt{WEEKDAY3.2}</td>
<td>Three-weekday intervals with Saturday and Sunday as weekend days (with the cycle three-weekday intervals aligned to Monday 4 Jan 1960)</td>
</tr>
<tr>
<td>\texttt{TENDAY4.2}</td>
<td>Four ten-day periods starting at the second TENDAY period</td>
</tr>
<tr>
<td>\texttt{SEMIMONTH2.2}</td>
<td>Intervals from the sixteenth of one month through the fifteenth of the next month</td>
</tr>
<tr>
<td>\texttt{MONTH2.2}</td>
<td>February-March, April-May, June-July, August-September, October-November, and December-January of the following year</td>
</tr>
<tr>
<td>\texttt{MONTH2}</td>
<td>January-February, March-April, May-June, July-August, September-October, November-December</td>
</tr>
<tr>
<td>\texttt{QTR3.2}</td>
<td>Three-month intervals starting on April 1, July 1, October 1, and January 1</td>
</tr>
<tr>
<td>\texttt{SEMIYEAR.3}</td>
<td>Six-month intervals, March-August and September-February</td>
</tr>
</tbody>
</table>
Interval Description

YEAR.10 Fiscal years starting in October
YEAR2.7 Biennial intervals starting in July of even years
YEAR2.19 Biennial intervals starting in July of odd years
YEAR4.11 Four-year intervals starting in November of leap years (frequency of U.S. presidential elections)
YEAR4.35 Four-year intervals starting in November of even years between leap years (frequency of U.S. midterm elections)
DTMONTH13 Thirteen-month intervals starting at midnight of January 1, 1960, such as November 1, 1957, December 1, 1958, January 1, 1960, February 1, 1961, and March 1, 1962
HOUR8.7 Eight-hour intervals starting at 6 AM, 2 PM, and 10 PM (might be used for work shifts)

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>qtr=intck(‘qtr’,'10jan95’d,’01jul95’d); put qtr;</td>
<td>2</td>
</tr>
<tr>
<td>year=intck(‘year’,'31dec94’d,’01jan95’d); put year;</td>
<td>1</td>
</tr>
<tr>
<td>year=intck(‘year’,'01jan94’d,’31dec94’d); put year;</td>
<td>0</td>
</tr>
<tr>
<td>semi=intck(‘semiyear’,'01jan95’d,’01jan98’d); put semi;</td>
<td>6</td>
</tr>
<tr>
<td>weekvar=intck(‘week2.2’,'01jan97’d,’31mar97’d); put weekvar;</td>
<td>6</td>
</tr>
<tr>
<td>wdvar=intck(‘weekday7w’,'01jan97’d,’01feb97’d); put wdvar;</td>
<td>26</td>
</tr>
<tr>
<td>y=’year’; date1=’1sep1991’d; date2=’1sep2001’d; newyears=intck(y,date1,date2); put newyears;</td>
<td>10</td>
</tr>
<tr>
<td>y=trim(’year ’); date1=’1sep1991’d + 300; date2=’1sep2001’d - 300; newyears=intck(y,date1,date2); put newyears;</td>
<td>8</td>
</tr>
</tbody>
</table>

In the second example, INTCK returns a value of 1 even though only one day has elapsed. This is because the interval from December 31, 1994 to January 1, 1995
contains the starting point for the YEAR interval. In the third example, however, a value of 0 is returned even though 364 days have elapsed. This is because the period between January 1, 1994 and December 31, 1994 does not contain the starting point for the interval.

In the fourth example, SAS returns a value of 6 because January 1, 1995 through January 1, 1998 contains six semiyearly intervals. (Note that if the ending date were December 31, 1997, SAS would count five intervals.) In the fifth example, SAS returns a value of 6 because there are six two-week intervals beginning on a first Monday during the period of January 1, 1997 through March 31, 1997. In the sixth example, SAS returns the value 26. That indicates that beginning with January 1, 1997, and counting only Saturdays as weekend days through February 1, 1997, the period contains 26 weekdays.

In the seventh example, the use of variables for the arguments is illustrated. The use of expressions for the arguments is illustrated in the last example.

**See Also**

Function:

“INTNX Function” on page 602

---

**INTNX Function**

Increments a date, time, or datetime value by a given interval or intervals, and returns a date, time, or datetime value

Category: Date and Time

**Syntax**

\[
\text{INTNX(} \text{interval<multiple><.shift-index>, start-from, increment<.alignment}> \text{)}
\]

**Arguments**

**interval**

specifies a character constant, a variable, or an expression that contains a time interval such as WEEK, SEMIYEAR, QTR, or HOUR.

**Requirement:** The type of interval (date, datetime, or time) must match the type of value in start-from and increment.

**See:** Table 4.4 on page 604 for a list of commonly used time intervals.

**multiple**

specifies a multiple of the interval. It sets the interval equal to a multiple of the interval type. For example, YEAR2 consists of two-year, or biennial, periods.

**See:** “Incrementing Dates and Times by Using Multipliers and by Shifting Intervals” on page 604 for more information.
shift-index
specifies the starting point of the interval. By default, the starting point is 1. A value that is greater than 1 shifts the start to a later point within the interval. The unit for shifting depends on the interval. For example, YEAR.3 specifies yearly periods that are shifted to start on the first of March of each calendar year and to end in February of the following year.

Restriction: The shift index cannot be greater than the number of periods in the entire interval. For example, YEAR2.24 has a valid shift index, but YEAR2.25 is invalid because there is no twenty-fifth month in a two-year interval.

Restriction: If the default shift period is the same as the interval type, then you can shift only multiperiod intervals with the shift index. For example, because MONTH type intervals shift by MONTH subperiods by default, you cannot shift monthly intervals with the shift index. However, you can shift bimonthly intervals with the shift index, because two MONTH intervals exist in each MONTH2 interval. The interval name MONTH2.2, for example, specifies bimonthly periods starting on the first day of even-numbered months.

See: “Incrementing Dates and Times by Using Multipliers and by Shifting Intervals” on page 604 for more information.

start-from
specifies a SAS expression that represents a SAS date, time, or datetime value that identifies a starting point.

increment
specifies a negative, positive, or zero integer that represents the number of date, time, or datetime intervals. Increment is the number of intervals to shift the value of start-from.

alignment
controls the position of SAS dates within the interval. Alignment can be one of these values:

BEGINNING
specifies that the returned date is aligned to the beginning of the interval.
Alias: B

MIDDLE
specifies that the returned date is aligned to the midpoint of the interval.
Alias: M

END
specifies that the returned date is aligned to the end of the interval.
Alias: E

SAMEDAY
specifies that the date that is returned is aligned to the same calendar date with the corresponding interval increment.
Alias: S
Alias: SAME
Default: BEGINNING
See: “Aligning SAS Date Output within Its Intervals” on page 605 for more information.

Details

The Basics The INTNX function increments a date, time, or datetime value by intervals such as DAY, WEEK, QTR, and MINUTE. The increment is based on a
starting date, time, or datetime value, and on the number of time intervals that you specify. The INTNX function returns the beginning date, time, or datetime value of the interval that you specify in the start–from argument. For example, to determine the date of the start of the week that is six weeks from the week of October 17, 2003, use the following example:

\[
\text{intnx('week', '17oct03'd, 6);}\]

INTNX returns the value 23NOV2003.

**Incrementing Dates and Times by Using Multipliers and by Shifting Intervals** SAS provides date, time, and datetime intervals for counting different periods of elapsed time. By using multipliers and shift indexes, you can create multiples of intervals and shift their starting point to construct more complex interval specifications.

The general form of an interval name is

\[
\text{name<multiplier><.shift-index>}
\]

Both the multiplier and the shift–index arguments are optional and default to 1. For example, YEAR, YEAR1, YEAR.1, and YEAR1.1 are all equivalent ways of specifying ordinary calendar years that begin in January. If you specify other values for multiplier and for shift–index, you can create multiple intervals that begin in different parts of the year. For example, the interval WEEK6.11 specifies six-week intervals starting on second Wednesdays.

**Understanding Time Intervals** Time intervals that do not nest within years or days are aligned relative to the SAS date or datetime value 0. SAS uses the arbitrary reference time of midnight on January 1, 1960, as the origin for non-shifted intervals. Shifted intervals are defined relative to January 1, 1960.

For example, MONTH13 defines the intervals January 1, 1960, February 1, 1961, March 1, 1962, and so on. The intervals December 1, 1958, November 1, 1957, October 1, 1956, and so on are dates that occurred before the base date of January 1, 1960.

As another example, the interval specification WEEK6.13 defines six-week periods starting on second Fridays. The convention of alignment relative to the period that contains January 1, 1960, determines where to start counting to determine which dates correspond to the second Fridays of six-week intervals.

The following table lists time intervals that are commonly used.

<table>
<thead>
<tr>
<th>Interval</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAY3</td>
<td>Three-day intervals starting on Sundays</td>
</tr>
<tr>
<td>WEEK</td>
<td>Weekly intervals starting on Sundays</td>
</tr>
<tr>
<td>WEEK.7</td>
<td>Weekly intervals starting on Saturdays</td>
</tr>
<tr>
<td>WEEK6.13</td>
<td>Six-week intervals starting on second Fridays</td>
</tr>
<tr>
<td>WEEK2</td>
<td>Biweekly intervals starting on first Sundays</td>
</tr>
<tr>
<td>WEEK1.1</td>
<td>Same as WEEK</td>
</tr>
<tr>
<td>WEEK.2</td>
<td>Weekly intervals starting on Mondays</td>
</tr>
<tr>
<td>WEEK6.3</td>
<td>Six-week intervals starting on first Tuesdays</td>
</tr>
<tr>
<td>WEEK6.11</td>
<td>Six-week intervals starting on second Wednesdays</td>
</tr>
<tr>
<td>WEEK4</td>
<td>Four-week intervals starting on first Sundays</td>
</tr>
<tr>
<td>WEEKDAY</td>
<td>Five-day work week with a Saturday-Sunday weekend</td>
</tr>
</tbody>
</table>
### Interval Descriptions

<table>
<thead>
<tr>
<th>Interval</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WEEKDAY1W</td>
<td>Six-day week with Sunday as a weekend day</td>
</tr>
<tr>
<td>WEEKDAY35W</td>
<td>Five-day week with Tuesday and Thursday as weekend days (W indicates day 3 and day 5 are weekend days)</td>
</tr>
<tr>
<td>WEEKDAY17W</td>
<td>Same as WEEKDAY</td>
</tr>
<tr>
<td>WEEKDAY67W</td>
<td>Five-day week with Friday and Saturday as weekend days</td>
</tr>
<tr>
<td>WEEKDAY3.2</td>
<td>Three-weekday intervals with Saturday and Sunday as weekend days (with the cycle three-weekday intervals aligned to Monday 4 Jan 1960)</td>
</tr>
<tr>
<td>TENDAY4.2</td>
<td>Four ten-day periods starting at the second TENDAY period</td>
</tr>
<tr>
<td>SEMIMONTH2.2</td>
<td>Intervals from the sixteenth of one month through the fifteenth of the next month</td>
</tr>
<tr>
<td>MONTH2.2</td>
<td>February–March, April–May, June–July, August–September, October–November, and December–January of the following year</td>
</tr>
<tr>
<td>MONTH2</td>
<td>January–February, March–April, May–June, July–August, September–October, November–December</td>
</tr>
<tr>
<td>QTR3.2</td>
<td>Three-month intervals starting on April 1, July 1, October 1, and January 1</td>
</tr>
<tr>
<td>SEMIYEAR.3</td>
<td>Six-month intervals, March–August and September–February</td>
</tr>
<tr>
<td>YEAR.10</td>
<td>Fiscal years starting in October</td>
</tr>
<tr>
<td>YEAR2.7</td>
<td>Biennial intervals starting in July of even years</td>
</tr>
<tr>
<td>YEAR2.19</td>
<td>Biennial intervals starting in July of odd years</td>
</tr>
<tr>
<td>YEAR4.11</td>
<td>Four-year intervals starting in November of leap years (frequency of U.S. presidential elections)</td>
</tr>
<tr>
<td>YEAR4.35</td>
<td>Four-year intervals starting in November of even years between leap years (frequency of U.S. midterm elections)</td>
</tr>
<tr>
<td>DTMONTH13</td>
<td>Thirteen-month intervals starting at midnight of January 1, 1960, such as November 1, 1957, December 1, 1958, January 1, 1960, February 1, 1961, and March 1, 1962</td>
</tr>
<tr>
<td>HOUR8.7</td>
<td>Eight-hour intervals starting at 6 a.m., 2 p.m., and 10 p.m. (might be used for work shifts)</td>
</tr>
</tbody>
</table>

For a complete list of the valid values for `interval`, see the “Intervals Used with Date and Time Functions” table in *SAS Language Reference: Concepts.*

### Aligning SAS Date Output within Its Intervals

SAS date values are normally aligned with the beginning of the time intervals that correspond to the interval type in `interval`. You can use the optional `alignment` argument to specify the alignment of the date that is returned. The values `BEGINNING`, `MIDDLE`, or `END` align the date to the beginning, middle, or end of the interval, respectively.
If you use the SAMEDAY value, then INTNX returns the same calendar date after computing the interval increment that you specified. The function automatically adjusts for the date if the date in the interval that is incremented does not exist. Here is an example:

```
intnx('month', '15mar2000'd, 5, 'sameday');  returns 15AUG2000
intnx('year', '29feb2000'd, 2, 'sameday');   returns 28FEB2002
intnx('month', '31aug2001'd, 1, 'sameday');  returns 30SEP2001
```

BEGINNING is the default value for alignment.

**Examples**

The following examples show how to use the INTNX function.

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>yr=intnx('year', '05feb94'd, 3);</td>
<td>13515</td>
</tr>
<tr>
<td>put yr / yr date7.;</td>
<td>01JAN97</td>
</tr>
<tr>
<td>x=intnx('month', '05jan95'd, 0);</td>
<td>12784</td>
</tr>
<tr>
<td>put x / x date7.;</td>
<td>01JAN95</td>
</tr>
<tr>
<td>next=intnx('semiyear', '01jan97'd, 1);</td>
<td>13696</td>
</tr>
<tr>
<td>put next / next date7.;</td>
<td>01JUL97</td>
</tr>
<tr>
<td>past=intnx('month2', '01aug96'd, -1);</td>
<td>13270</td>
</tr>
<tr>
<td>put past / past date7.;</td>
<td>01MAY96</td>
</tr>
<tr>
<td>sm=intnx('semimonth2.2', '01apr97'd, 4);</td>
<td>13711</td>
</tr>
<tr>
<td>put sm / sm date7.;</td>
<td>16JUL97</td>
</tr>
<tr>
<td>x='month';</td>
<td></td>
</tr>
<tr>
<td>date='1jun1990'd;</td>
<td></td>
</tr>
<tr>
<td>nextmon=intnx(x, date, 1);</td>
<td>11139</td>
</tr>
<tr>
<td>put nextmon / nextmon date7.;</td>
<td>01JUL90</td>
</tr>
<tr>
<td>x1='month ';</td>
<td></td>
</tr>
<tr>
<td>x2=trim(x1);</td>
<td></td>
</tr>
<tr>
<td>date='1jun1990'd - 100;</td>
<td></td>
</tr>
<tr>
<td>nextmonth=intnx(x2, date, 1);</td>
<td>11017</td>
</tr>
<tr>
<td>put nextmonth / nextmonth date7.;</td>
<td>01MAR90</td>
</tr>
</tbody>
</table>

The following examples show the results of advancing a date by using the optional alignment argument.

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>date1=intnx('month', '01jan95'd, 5, 'beginning');</td>
<td>12935</td>
</tr>
<tr>
<td>put date1 / date1 date7.;</td>
<td>01JUN95</td>
</tr>
<tr>
<td>date2=intnx('month', '01jan95'd, 5, 'middle');</td>
<td>12949</td>
</tr>
<tr>
<td>put date2 / date2 date7.;</td>
<td>15JUN95</td>
</tr>
<tr>
<td>date3=intnx('month', '01jan95'd, 5, 'end');</td>
<td>12964</td>
</tr>
<tr>
<td>put date3 / date3 date7.;</td>
<td>30JUN95</td>
</tr>
<tr>
<td>date4=intnx('month', '01jan95'd, 5, 'sameday');</td>
<td>12935</td>
</tr>
<tr>
<td>put date4 / date4 date7.;</td>
<td>01JUN95</td>
</tr>
</tbody>
</table>
## Functions and CALL Routines

### INTRR Function

**Returns the internal rate of return as a fraction**

**Category:** Financial

### Syntax

\[ \text{INTRR}(\text{freq}, c_0, c_1, \ldots, c_n) \]

### Arguments

- **\( freq \)**
  
  is numeric, the number of payments over a specified base period of time that is associated with the desired internal rate of return.

  **Range:** \( freq > 0 \)

  **Exception:** The case \( freq = 0 \) is a flag to allow continuous compounding.

- **\( c_0, c_1, \ldots, c_n \)**

  are numeric, the optional cash payments.

### Details

The INTRR function returns the internal rate of return over a specified base period of time for the set of cash payments \( c_0, c_1, \ldots, c_n \). The time intervals between any two
consecutive payments are assumed to be equal. The argument \( freq > 0 \) describes the number of payments that occur over the specified base period of time.

The internal rate of return is the interest rate such that the sequence of payments has a 0 net present value (see the NETPV function). It is given by

\[
 r = \begin{cases} 
 \frac{1}{x} & f r e q > 0 \\
 -\log_e (x) & f r e q = 0 
\end{cases}
\]

where \( x \) is the real root, nearest to 1, of the polynomial

\[
\sum_{i=0}^{n} c_i x^i = 0
\]

The routine uses Newton’s method to look for the internal rate of return nearest to 0. Depending on the value of payments, a root for the equation does not always exist; in that case, a missing value is returned.

Missing values in the payments are treated as 0 values. When \( freq > 0 \), the computed rate of return is the effective rate over the specified base period. To compute a quarterly internal rate of return (the base period is three months) with monthly payments, set \( freq \) to 3.

If \( freq \) is 0, continuous compounding is assumed and the base period is the time interval between two consecutive payments. The computed internal rate of return is the nominal rate of return over the base period. To compute with continuous compounding and monthly payments, set \( freq \) to 0. The computed internal rate of return will be a monthly rate.

**Examples**

For an initial outlay of $400 and expected payments of $100, $200, and $300 over the following three years, the annual internal rate of return can be expressed as

\[
rate=intrr(1,-400,100,200,300);
\]

The value returned is 0.19438.

---

**INTZ Function**

Returns the integer portion of the argument, using zero fuzzing

**Category:** Truncation

**Syntax**

\[
INTZ \ (argument)
\]
Arguments

argument

is a numeric constant, variable, or expression.

Details

The following rules apply:

- If the value of the argument is an exact integer, INTZ returns that integer.
- If the argument is positive and not an integer, INTZ returns the largest integer that is less than the argument.
- If the argument is negative and not an integer, INTZ returns the smallest integer that is greater than the argument.

Comparisons

Unlike the INT function, the INTZ function uses zero fuzzing. If the argument is within 1E-12 of an integer, the INT function fuzzes the result to be equal to that integer. The INTZ function does not fuzz the result. Therefore, with the INTZ function you might get unexpected results.

Examples

The following SAS statements produce these results.

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>var1=2.1;</td>
<td></td>
</tr>
<tr>
<td>a=intz(var1);</td>
<td></td>
</tr>
<tr>
<td>put a;</td>
<td>2</td>
</tr>
<tr>
<td>var2=-2.4;</td>
<td></td>
</tr>
<tr>
<td>b=intz(var2);</td>
<td></td>
</tr>
<tr>
<td>put b;</td>
<td>-2</td>
</tr>
<tr>
<td>var3=1+1.e-11;</td>
<td></td>
</tr>
<tr>
<td>c=intz(var3);</td>
<td></td>
</tr>
<tr>
<td>put c;</td>
<td>1</td>
</tr>
<tr>
<td>f=intz(-1.6);</td>
<td></td>
</tr>
<tr>
<td>put f;</td>
<td>-1</td>
</tr>
</tbody>
</table>

See Also

Functions:

“CEIL Function” on page 429
“CEILZ Function” on page 430
“FLOOR Function” on page 544
IORCMSG Function

Returns a formatted error message for _IORC_

Category: SAS File I/O

Syntax

character-variable=IORCMSG()

Arguments

character-variable specifies a character variable.

Tip: If the length has been previously defined, then the result will be truncated or padded as needed.

Default: The default length is 200 characters.

Details

If the IORCMSG function returns a value to a variable that has not yet been assigned a length, then by default the variable is assigned a length of 200.

The IORCMSG function returns the formatted error message that is associated with the current value of the automatic variable _IORC_. The _IORC_ variable is created when you use the MODIFY statement, or when you use the SET statement with the KEY= option. The value of the _IORC_ variable is internal and is meant to be read in conjunction with the SYSRC autocall macro. If you try to set _IORC_ to a specific value, you might get unexpected results.

Examples

In the following program, observations are either rewritten or added to the updated master file that contains bank accounts and current bank balance. The program queries the _IORC_ variable and returns a formatted error message if the _IORC_ value is unexpected.

```sas
libname bank ‘SAS-data-library’;

data bank.master(index=(AccountNum));
   infile ‘external-file-1’;
```
format balance dollar8.;
input @ 1 AccountNum $ 1--3 @ 5 balance 5--9;
run;

data bank.trans(index=(AccountNum));
infile 'external-file-2';
format deposit dollar8.;
input @ 1 AccountNum $ 1--3 @ 5 deposit 5--9;
run;

data bank.master;
set bank.trans;
modify bank.master key=AccountNum;
if (_IORC_ EQ %sysrc(_SOK)) then
   do;
      balance=balance+deposit;
      replace;
   end;
else
   if (_IORC_ = %sysrc(_DSENOM)) then
      do;
         balance=deposit;
         output;
         _error_ = 0;
      end;
   else
      do;
         errmsg=IORCMSG();
         put 'Unknown error condition:'
         errmsg;
      end;
run;

---

**IQR Function**

**Returns the interquartile range**

**Category:** Descriptive Statistics

---

**Syntax**

\[ \text{IQR}(\text{value-1} \ , \text{value-2} \ldots) \]

**Arguments**

\[ \text{value} \]

specifies a numeric constant, variable, or expression of which the interquartile range is to be computed.
Details

If all arguments have missing values, the result is a missing value. Otherwise, the result is the interquartile range of the non-missing values. The formula for the interquartile range is the same as the one that is used in the UNIVARIATE procedure. For more information, see Base SAS Procedures Guide.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>iqr=iqr(2,4,1,3,999999); put iqr;</td>
<td>2</td>
</tr>
</tbody>
</table>

See Also

Functions:
“MAD Function” on page 641
“PCTL Function” on page 694

IRR Function

Returns the internal rate of return as a percentage

Category: Financial

Syntax

IRR(freq,c0,c1,...,cn)

Arguments

freq
is numeric, the number of payments over a specified base period of time that is associated with the desired internal rate of return.

Range: freq > 0.

Exception: The case freq = 0 is a flag to allow continuous compounding.

c0,c1,...,cn
are numeric, the optional cash payments.

Comparisons

The IRR function is identical to INTRRR, except that the rate returned is a percentage.
**JBESSEL Function**

Returns the value of the bessel function

**Category:** Mathematical

**Syntax**

\[
\text{JBESSEL}(\nu, x)
\]

**Arguments**

\(\nu\)

is numeric.

**Range:** \(\nu \geq 0\)

\(x\)

is numeric.

**Range:** \(x \geq 0\)

**Details**

The JBESSEL function returns the value of the bessel function of order \(\nu\) evaluated at \(x\) (For more information, see Abramowitz and Stegun 1964; Amos, Daniel, and Weston 1977).

**Examples**

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x = \text{jbessel}(2, 2));</td>
<td>0.3528340286</td>
</tr>
</tbody>
</table>

**JULDATE Function**

Returns the Julian date from a SAS date value

**Category:** Date and Time

**Syntax**

\[
\text{JULDATE}(\text{date})
\]
Arguments

*date*
specifies a SAS date value.

Details

The JULDATE function converts a SAS date value to a five- or seven-digit Julian date. If *date* falls within the 100-year span defined by the system option YEARCUTOFF=, the result has five digits: the first two digits represent the year, and the next three digits represent the day of the year (1 to 365, or 1 to 366 for leap years). Otherwise, the result has seven digits: the first four digits represent the year, and the next three digits represent the day of the year. For example, if YEARCUTOFF=1920, JULDATE would return 97001 for January 1, 1997, and return 1878365 for December 31, 1878.

Comparisons

The function JULDATE7 is similar to JULDATE except that JULDATE7 always returns a four digit year. Thus JULDATE7 is year 2000 compliant because it eliminates the need to consider the implications of a two digit year.

Examples

The following SAS statements produce these results:

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>julian=juldate('31dec99'd);</td>
<td>99365</td>
</tr>
<tr>
<td>julian=juldate('01jan2099'd);</td>
<td>2099001</td>
</tr>
</tbody>
</table>

See Also

Function:

“DATEJUL Function” on page 479
“JULDATE7 Function” on page 614

System Option:

“YEARCUTOFF= System Option” on page 1630

JULDATE7 Function

Returns a seven-digit Julian date from a SAS date value

Category: Date and Time
Syntax

\texttt{JULDATE7(date)}

Arguments

date
specifies a SAS date value.

Details

The JULDATE7 function returns a seven digit Julian date from a SAS date value. The first four digits represent the year, and the next three digits represent the day of the year.

Comparisons

The function JULDATE7 is similar to JULDATE except that JULDATE7 always returns a four digit year. Thus JULDATE7 is year 2000 compliant because it eliminates the need to consider the implications of a two digit year.

Examples

The following SAS statements produce these results:

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>julian=juldate7('31dec96'd);</td>
<td>1996366</td>
</tr>
<tr>
<td>julian=juldate7('01jan2099'd);</td>
<td>2099001</td>
</tr>
</tbody>
</table>

See Also

Function:

“JULDATE Function” on page 613

KCOMPARE Function

Returns the result of a comparison of character strings

Category: DBCS


KCOMPRESS Function

Removes specific characters from a character string
KCOUNT Function

Returns the number of double-byte characters in a string
Category: DBCS

KCVT Function

Converts data from an encoding code to another encoding code
Category: DBCS

KINDEX Function

Searches a character expression for a string of characters
Category: DBCS

KINDEXC Function

Searches a character expression for specific characters
Category: DBCS

KLEFT Function

Left aligns a character expression by removing unnecessary leading DBCS blanks and SO/SI
KLENGTH Function

Returns the length of an argument

Category: DBCS

KLOWCASE Function

Converts all letters in an argument to lowercase

Category: DBCS

KREVERSE Function

Reverses a character expression

Category: DBCS

KRIGHT Function

Right aligns a character expression by trimming trailing DBCS blanks and SO/SI

Category: DBCS

KSCAN Function

Selects a specific word from a character expression
KSTRCAT Function

Concatenates two or more character strings

Category: DBCS

KSUBSTR Function

Extracts a substring from an argument

Category: DBCS

KSUBSTRB Function

Extracts a substring from an argument according to the byte position of the substring in the argument

Category: DBCS

KTRANSLATE Function

Replaces specific characters in a character expression

Category: DBCS

KTRIM Function

Removes trailing DBCS blanks and SO/SI from character expressions
Functions and CALL Routines

KURTOSIS Function

Returns the kurtosis

KTRUNCATE Function

Truncates a numeric value to a specified length

KUPCASE Function

Converts all single-byte letters in an argument to uppercase

KUPDATE Function

Inserts, deletes, and replaces character value contents

KUPDATEB Function

Inserts, deletes, and replaces the contents of the character value according to the byte position of the character value in the argument

Category: DBCS

Category: DBCS

Category: DBCS

Category: DBCS
Category: Descriptive Statistics

Syntax

KURTOSIS(argument, argument, ...)

Arguments

argument
is numeric. At least four nonmissing arguments are required. Otherwise, the function returns a missing value. The argument list can consist of a variable list, which is preceded by OF.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1=kurtosis(5,9,3,6);</td>
<td>0.928</td>
</tr>
<tr>
<td>x2=kurtosis(5,8,9,6,..);</td>
<td>-3.3</td>
</tr>
<tr>
<td>x3=kurtosis(8,9,6,1);</td>
<td>1.5</td>
</tr>
<tr>
<td>x4=kurtosis(8,1,6,1);</td>
<td>-4.483379501</td>
</tr>
<tr>
<td>x5=kurtosis(of x1-x4);</td>
<td>-5.065692754</td>
</tr>
</tbody>
</table>

KVERIFY Function

Returns the position of the first character that is unique to an expression

Category: DBCS


LAG Function

Returns values from a queue

Category: Special

Syntax

LAG<n>(argument)
Arguments

\[ n \]

specifies the number of lagged values.

\textit{argument}

is numeric or character.

Details

If the LAG function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

The LAG functions, \texttt{LAG1, LAG2, ...}, \texttt{LAG100} return values from a queue. \texttt{LAG1} can also be written as \texttt{LAG}. A \texttt{LAGn} function stores a value in a queue and returns a value stored previously in that queue. Each occurrence of a \texttt{LAGn} function in a program generates its own queue of values.

The queue for \texttt{LAGn} is initialized with \[ n \] missing values, where \[ n \] is the length of the queue (for example, a \texttt{LAG2} queue is initialized with two missing values). When \texttt{LAGn} is executed, the value at the top of the queue is removed and returned, the remaining values are shifted upwards, and the new value of the argument is placed at the bottom of the queue. Hence, missing values are returned for the first \[ n \] executions of \texttt{LAGn}, after which the lagged values of the argument begin to appear.

\textit{Note:} Storing values at the bottom of the queue and returning values from the top of the queue occurs only when the function is executed. A \texttt{LAGn} function that is executed conditionally will store and return values only from the observations for which the condition is satisfied. See Example 2 on page 622.

If the argument of \texttt{LAGn} is an array name, a separate queue is maintained for each variable in the array.

Examples

**Example 1: Creating a Data Set**

The following program creates a data set that contains the values for \textit{X}, \textit{Y}, and \textit{Z}.

\begin{verbatim}
options pagesize=25 linesize=64 nodate pageno=1;

data one;
   input X @@;
   Y=lag1(x);
   Z=lag2(x);
datalines;
1 2 3 4 5 6
;
proc print;
   title 'Lag Output';
run;
\end{verbatim}
LAG1 returns one missing value and the values of X (lagged once). LAG2 returns two missing values and the values of X (lagged twice).

**Example 2: Storing Every Other Lagged Value**  This example shows the difference in output when you use conditional and unconditional logic in your program. Because the LAG function stores values on the queue only when it is called, you must call LAG unconditionally to get the correct answers.

```sas
options pagesize=25 linesize=64 nodate pageno=1;

title 'Store Every Other Lagged Value';

data test;
  input x @@;
  if mod(x,2)=0 then a=lag(x);
  b=lag(x);
  if mod(x,2)=0 then c=b;
  label a='(WRONG) a' c='(RIGHT) c';
  datalines;
  1 2 3 4 5 6 7 8
;
proc print label data=test;
run;
```

See Also

Function:

“DIF Function” on page 496
LARGEST Function

Returns the $k$th largest non-missing value

Category: Descriptive Statistics

Syntax

\[
\text{LARGEST} (k, \text{value-1}<, \text{value-2} ...>)
\]

Arguments

\[k\]

is a numeric constant, variable, or expression that specifies which value to return.

\[value\]

specifies the value of a numeric constant, variable, or expression to be processed.

Details

If $k$ is missing, less than zero, or greater than the number of values, the result is a missing value and _ERROR_ is set to 1. Otherwise, if $k$ is greater than the number of non-missing values, the result is a missing value but _ERROR_ is not set to 1.

Examples

The following SAS statements produce these results.

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
</table>
| $k=1$; \[
\text{largest1=largest}(k, 456, 789, .Q, 123); 
\text{put largest1;} 
\] | 789 |
| $k=2$; \[
\text{larget2=largest}(k, 456, 789, .Q, 123); 
\text{put largest2;} 
\] | 456 |
| $k=3$; \[
\text{largest3=largest}(k, 456, 789, .Q, 123); 
\text{put largest3;} 
\] | 123 |
| $k=4$; \[
\text{largest4=largest}(k, 456, 789, .Q, 123); 
\text{put largest4;} 
\] | . |

See Also

Functions:

“ORDINAL Function” on page 692
LBOUND Function

Returns the lower bound of an array

Category: Array

Syntax

LBOUND<n>(array-name)
LBOUND(array-name, bound-n)

Arguments

n
specifies the dimension for which you want to know the lower bound. If no n value is specified, the LBOUND function returns the lower bound of the first dimension of the array.

array-name
specifies the name of an array defined previously in the same DATA step.

bound-n
specifies the dimension for which you want to know the lower bound. Use bound-n only if n is not specified.

Details

The LBOUND function returns the lower bound of a one-dimensional array or the lower bound of a specified dimension of a multidimensional array. Use LBOUND in array processing to avoid changing the lower bound of an iterative DO group each time you change the bounds of the array. LBOUND and HBOUND can be used together to return the values of the lower and upper bounds of an array dimension.

Examples

Example 1: One-dimensional Array
In this example, LBOUND returns the lower bound of the dimension, a value of 2. SAS repeats the statements in the DO loop five times.

```sas
array big{2:6} weight sex height state city;
do i=lbound(big) to hbound(big);
   ...more SAS statements...;
end;
```

Example 2: Multidimensional Array
This example shows two ways of specifying the LBOUND function for multidimensional arrays. Both methods return the same value for LBOUND, as shown in the table that follows the SAS code example.
array mult{2:6,4:13,2} mult1-mult100;

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Alternative Syntax</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBOUND(MULT)</td>
<td>LBOUND(MULT,1)</td>
<td>2</td>
</tr>
<tr>
<td>LBOUND2(MULT)</td>
<td>LBOUND(MULT,2)</td>
<td>4</td>
</tr>
<tr>
<td>LBOUND3(MULT)</td>
<td>LBOUND(MULT,3)</td>
<td>1</td>
</tr>
</tbody>
</table>

See Also

Functions:
   “DIM Function” on page 498
   “HBOUND Function” on page 576

Statements:
   “ARRAY Statement” on page 1102
   “Array Reference Statement” on page 1107
   “Array Processing” in SAS Language Reference: Concepts

LEFT Function

Left aligns a SAS character expression

Category: Character

Syntax

LEFT(argument)

Arguments

argument
   specifies any SAS character expression.

Details

If the LEFT function returns a value to a variable that has not yet been assigned a length, by default the variable length is determined by the length of the first argument. LEFT returns an argument with leading blanks moved to the end of the value. The argument's length does not change.
Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=' DUE DATE';</td>
<td></td>
</tr>
<tr>
<td>b=left(a);</td>
<td></td>
</tr>
<tr>
<td>put b;</td>
<td>DUE DATE</td>
</tr>
</tbody>
</table>

See Also

Functions:
- “COMPRESS Function” on page 456
- “RIGHT Function” on page 788
- “TRIM Function” on page 863

LENGTH Function

Returns the length of a non-blank character string, excluding trailing blanks, and returns 1 for a blank character string

Category: Character

Syntax

LENGTH(string)

Arguments

string
specifies a character constant, variable, or expression.

Details

The LENGTH function returns an integer that represents the position of the rightmost non-blank character in string. If the value of string is blank or missing, LENGTH returns a value of 1. If string is a numeric variable (either initialized or uninitialized), LENGTH returns a value of 12 and prints a note in the SAS log that the numeric values have been converted to character values.

Comparisons

- The LENGTH and LENGTHN functions return the same value for non-blank character strings. LENGTH returns a value of 1 for blank character strings, whereas LENGTHN returns a value of 0.
The LENGTH function returns the length of a character string, excluding trailing blanks, whereas the LENGTHC function returns the length of a character string, including trailing blanks. LENGTH always returns a value that is less than or equal to the value returned by LENGTHC.

The LENGTH function returns the length of a character string, excluding trailing blanks, whereas the LENGTHM function returns the amount of memory in bytes that is allocated for a character string. LENGTH always returns a value that is less than or equal to the value returned by LENGTHM.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>len=length('ABCDEF');</code></td>
<td>6</td>
</tr>
<tr>
<td><code>put len;</code></td>
<td></td>
</tr>
<tr>
<td><code>len2=length(' ');</code></td>
<td>1</td>
</tr>
<tr>
<td><code>put len2;</code></td>
<td></td>
</tr>
</tbody>
</table>

See Also

Functions:
- “LENGTHC Function” on page 627
- “LENGTHM Function” on page 628
- “LENGTHN Function” on page 630

LENGTHC Function

Returns the length of a character string, including trailing blanks

Category: Character

Syntax

LENGTHC(string)

Arguments

string

specifies a character constant, variable, or expression.

Details

The LENGTHC function returns an integer that represents the position of the rightmost blank or non-blank character in string. If the value of string is missing and contains
blanks, LENGTHC returns the number of blanks in \textit{string}. If the value of \textit{string} is missing and contains no blanks, LENGTHC returns a value of 0. If \textit{string} is a numeric variable (either initialized or uninitialized), LENGTHC returns a value of 12 and prints a note in the SAS log that the numeric values have been converted to character values.

**Comparisons**

- The LENGTHC function returns the length of a character string, including trailing blanks, whereas the LENGTH and LENGTHN functions return the length of a character string, excluding trailing blanks. LENGTHC always returns a value that is greater than or equal to the values returned by LENGTH and LENGTHN.

- The LENGTHC function returns the length of a character string, including trailing blanks, whereas the LENGTHM function returns the amount of memory in bytes that is allocated for a character string. For fixed-length character strings, LENGTHC and LENGTHM always return the same value. For varying-length character strings, LENGTHC always returns a value that is less than or equal to the value returned by LENGTHM.

**Examples**

The following SAS statements produce these results:

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=lengthc('variable with trailing blanks '); put x;</td>
<td>32</td>
</tr>
<tr>
<td>length fixed $35; fixed='variable with trailing blanks '; x=lengthc(fixed); put x;</td>
<td>35</td>
</tr>
</tbody>
</table>

**See Also**

Functions:

- “LENGTH Function” on page 626
- “LENGTHM Function” on page 628
- “LENGTHN Function” on page 630

**LENGTHM Function**

Returns the amount of memory (in bytes) that is allocated for a character string

**Category:** Character

**Syntax**

\texttt{LENGTHM(string)}
Arguments

*string*

specifies a character constant, variable, or expression.

Details

The `LENGTHM` function returns an integer that represents the amount of memory in bytes that is allocated for `string`. If `string` is a numeric variable (either initialized or uninitialized), `LENGTHM` returns a value of 12 and prints a note in the SAS log that the numeric values have been converted to character values.

Comparisons

The `LENGTHM` function returns the amount of memory in bytes that is allocated for a character string, whereas the `LENGTH`, `LENGTHC`, and `LENGTHN` functions return the length of a character string. `LENGTHM` always returns a value that is greater than or equal to the values returned by `LENGTH`, `LENGTHC`, and `LENGTHN`.

Examples

**Example 1: Determining the Amount of Allocated Memory for a Variable with Trailing Blanks**

This example determines the amount of memory (in bytes) that is allocated to a variable with trailing blanks.

```sas
data _null_;  
x=lengthm('variable with trailing blanks ');  
put x;  
run;
```

The following line is written to the SAS log:

```
32
```

**Example 2: Determining the Amount of Allocated Memory for a Variable from an External File**

This example determines the amount of memory (in bytes) that is allocated to a variable that is input into a SAS file from an external file.

```sas
data _null_;  
   file 'test.txt';  
   put 'trailing blanks ';  
run;
```

```sas
data test;  
   infile 'test.txt';  
   input;  
   x=lengthm(_infile_);  
   put x;  
run;
```

The following line is written to the SAS log:

```
256
```
See Also

Functions:

“LENGTH Function” on page 626
“LENGTHC Function” on page 627
“LENGTHN Function” on page 630

LENGTHN Function

Returns the length of a non-blank character string, excluding trailing blanks, and returns 0 for a blank character string

Category: Character

Syntax

**LENGTHN**(string)

Arguments

**string**

specifies a character constant, variable, or expression.

Details

The LENGTHN function returns an integer that represents the position of the rightmost non-blank character in string. If the value of string is blank or missing, LENGTHN returns a value of 0. If string is a numeric variable (either initialized or uninitialized), LENGTHN returns a value of 12 and prints a note in the SAS log that the numeric values have been converted to character values.

Comparisons

- The LENGTHN and LENGTH functions return the same value for non-blank character strings. LENGTHN returns a value of 0 for blank character strings, whereas LENGTH returns a value of 1.
- The LENGTHN function returns the length of a character string, excluding trailing blanks, whereas the LENGTHC function returns the length of a character string, including trailing blanks. LENGTHN always returns a value that is less than or equal to the value returned by LENGTHC.
- The LENGTHN function returns the length of a character string, excluding trailing blanks, whereas the LENGTHM function returns the amount of memory in bytes that is allocated for a character string. LENGTHN always returns a value that is less than or equal to the value returned by LENGTHM.
Examples

SAS Statements | Results
---|---
`len=lengthn('ABCDEF');`<br>`put len;`<br>`len2=lengthn(' ');`<br>`put len2;` | 6<br>0

See Also

Functions:
- “LENGTH Function” on page 626
- “LENGTHC Function” on page 627
- “LENGTHM Function” on page 628

LGAMMA Function

Returns the natural logarithm of the Gamma function

Category: Mathematical

Syntax

`LGAMMA(argument)`

Arguments

`argument` is numeric.

Range: must be positive.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x=lgamma(2);</code></td>
<td>0</td>
</tr>
<tr>
<td><code>x=lgamma(1.5);</code></td>
<td>-0.120782238</td>
</tr>
</tbody>
</table>
LIBNAME Function

Assigns or deassigns a libref for a SAS data library

Category: SAS File I/O
See: LIBNAME Function in the documentation for your operating environment.

Syntax

LIBNAME(libref,<SAS-data-library>,<engine>,<options>>)>

Arguments

libref
specifies the libref that is assigned to a SAS data library.

SAS-data-library
specifies the physical name of the SAS data library that is associated with the libref. Specify this name as required by the host operating environment.

engine
specifies the engine that is used to access SAS files opened in the data library. If you are specifying a SAS/SHARE server, then the engine should be REMOTE.

options	names one or more options honored by the specified engine, delimited with blanks.

Details

If the LIBNAME function returns 0, then the function was successful. However, you could receive a non-zero value, even if the function was successful. A non-zero value is returned if an error, warning, or note is produced. To determine if the function was successful, look through the SAS log and use the following guidelines:

- If a warning or note was generated, then the function was successful.
- If an error was generated, then the function was not successful.

Operating Environment Information: Some systems allow a SAS-data-library value of ' ' (with a space) to assign a libref to the current directory. The behavior of LIBNAME when a single space is specified for SAS-data-library is host dependent.

If no value is provided for SAS-data-library or if SAS-data-library has a value of " (with no space), LIBNAME disassociates the libref from the data library.

Under some operating environments, the user can assign librefs using system commands outside the SAS session.

Examples

Example 1: Assigning a Libref

This example attempts to assign the libref NEW to the SAS data library MYLIB. If an error or warning occurs, the message is written to the SAS log. Note that in a macro statement you do not enclose character strings in quotation marks.
Example 2: Deassigning a Libref

This example deassigns the libref NEW that has been previously associated with the data library MYLIB in the preceding example. If an error or warning occurs, the message is written to the SAS log. Note that in a macro statement you do not enclose character strings in quotation marks.

```
%if (libname(new)) %then
  %put %sysfunc(sysmsg());
```

See Also

Function:

“LIBREF Function” on page 633

---

**LIBREF Function**

Verifies that a libref has been assigned

Category: SAS File I/O

See: LIBREF Function in the documentation for your operating environment.

**Syntax**

```
LIBREF(libref)
```

**Arguments**

`libref`

specifies the libref to be verified.

**Details**

LIBREF returns 0 if the operation was successful, ≠0 if it was not successful.

**Examples**

This example verifies a libref. If an error or warning occurs, the message is written to the log. Under some operating environments, the user can assign librefs by using system commands outside the SAS session.

```
%if (%sysfunc(libname(new,MYLIB))) %then
  %put %sysfunc(sysmsg());
```
See Also

Function:
“LIBNAME Function” on page 632

LOG Function

Returns the natural (base e) logarithm

Category: Mathematical

Syntax

LOG(argument)

Arguments

argument is numeric.

Range: must be positive.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=log(1.0);</td>
<td>0</td>
</tr>
<tr>
<td>x=log(10.0);</td>
<td>2.302585093</td>
</tr>
</tbody>
</table>

LOG10 Function

Returns the logarithm to the base 10

Category: Mathematical

Syntax

LOG10(argument)
Arguments

*argument* is numeric.

**Range:** must be positive.

**Examples**

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x=log10(1.0);</code></td>
<td>0</td>
</tr>
<tr>
<td><code>x=log10(10.0);</code></td>
<td>1</td>
</tr>
<tr>
<td><code>x=log10(100.0);</code></td>
<td>2</td>
</tr>
</tbody>
</table>

**LOG2 Function**

Returns the logarithm to the base 2

**Category:** Mathematical

**Syntax**

`LOG2(argument)`

**Arguments**

*argument* is numeric.

**Range:** must be positive.

**Examples**

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x=log2(2.0);</code></td>
<td>1</td>
</tr>
<tr>
<td><code>x=log2(0.5);</code></td>
<td>-1</td>
</tr>
</tbody>
</table>

**LOGBETA Function**

Returns the logarithm of the beta function
Category: Mathematical

Syntax

LOGBETA(a,b)

Arguments

$\alpha$

is the first shape parameter, where $\alpha > 0$.

$\beta$

is the second shape parameter, where $\beta > 0$.

Details

The LOGBETA function is mathematically given by the equation

$$\log(\beta(a, \beta)) = \log(\Gamma(a)) + \log(\Gamma(b)) - \log(\Gamma(a + b))$$

where $\Gamma(.)$ is the gamma function.

If the expression cannot be computed, LOGBETA returns a missing value.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOGBETA(5,3);</td>
<td>-4.653960350</td>
</tr>
</tbody>
</table>

See Also

Function:

“BETA Function” on page 329

LOGCDF Function

Computes the logarithm of a left cumulative distribution function

Category: Probability

See: “CDF Function” on page 416
Syntax

LOGCDF('dist', quantile, parm-1,...,parm-k)

Arguments

'dist'
is a character string that identifies the distribution. Valid distributions are as follows:

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bernoulli</td>
<td>'BERNOULLI'</td>
</tr>
<tr>
<td>Beta</td>
<td>'BETA'</td>
</tr>
<tr>
<td>Binomial</td>
<td>'BINOMIAL'</td>
</tr>
<tr>
<td>Cauchy</td>
<td>'CAUCHY'</td>
</tr>
<tr>
<td>Chi-Square</td>
<td>'CHISQUARE'</td>
</tr>
<tr>
<td>Exponential</td>
<td>'EXPONENTIAL'</td>
</tr>
<tr>
<td>F</td>
<td>'F'</td>
</tr>
<tr>
<td>Gamma</td>
<td>'GAMMA'</td>
</tr>
<tr>
<td>Geometric</td>
<td>'GEOMETRIC'</td>
</tr>
<tr>
<td>Hypergeometric</td>
<td>'HYPERGEOMETRIC'</td>
</tr>
<tr>
<td>Laplace</td>
<td>'LAPLACE'</td>
</tr>
<tr>
<td>Logistic</td>
<td>'LOGISTIC'</td>
</tr>
<tr>
<td>Lognormal</td>
<td>'LOGNORMAL'</td>
</tr>
<tr>
<td>Negative binomial</td>
<td>'NEGBINOMIAL'</td>
</tr>
<tr>
<td>Normal</td>
<td>'NORMAL'</td>
</tr>
<tr>
<td>Normal mixture</td>
<td>'NORMALMIX'</td>
</tr>
<tr>
<td>Pareto</td>
<td>'PARETO'</td>
</tr>
<tr>
<td>Poisson</td>
<td>'POISSON'</td>
</tr>
<tr>
<td>T</td>
<td>'T'</td>
</tr>
<tr>
<td>Uniform</td>
<td>'UNIFORM'</td>
</tr>
<tr>
<td>Wald (inverse Gaussian)</td>
<td>'WALD'</td>
</tr>
<tr>
<td>Weibull</td>
<td>'WEIBULL'</td>
</tr>
</tbody>
</table>

*Note:* Except for T, F, and NORMALMIX, you can minimally identify any distribution by its first four characters.

quantile
is a numeric random variable.

parm-1,...,parm-k
are optional shape, location, or scale parameters appropriate for the specific distribution.
The LOGCDF function computes the logarithm of a left cumulative distribution function (logarithm of the left side) from various continuous and discrete distributions. For more information, see the “CDF Function” on page 416.

LOGPDF Function

Computes the logarithm of a probability density (mass) function

Category: Probability

Alias: LOGPMF

See: “PDF Function” on page 695

Syntax

LOGPDF('dist', quantile, parm-1,...,parm-k)

Arguments

'dist'

is a character string that identifies the distribution. Valid distributions are as follows:

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bernoulli</td>
<td>‘BERNOULLI’</td>
</tr>
<tr>
<td>Beta</td>
<td>‘BETA’</td>
</tr>
<tr>
<td>Binomial</td>
<td>‘BINOMIAL’</td>
</tr>
<tr>
<td>Cauchy</td>
<td>‘CAUCHY’</td>
</tr>
<tr>
<td>Chi-Square</td>
<td>‘CHISQUARE’</td>
</tr>
<tr>
<td>Exponential</td>
<td>‘EXPONENTIAL’</td>
</tr>
<tr>
<td>F</td>
<td>‘F’</td>
</tr>
<tr>
<td>Gamma</td>
<td>‘GAMMA’</td>
</tr>
<tr>
<td>Geometric</td>
<td>‘GEOMETRIC’</td>
</tr>
<tr>
<td>Hypergeometric</td>
<td>‘HYPERGEOMETRIC’</td>
</tr>
<tr>
<td>Laplace</td>
<td>‘LAPLAWE’</td>
</tr>
<tr>
<td>Logistic</td>
<td>‘LOGISTIC’</td>
</tr>
<tr>
<td>Lognormal</td>
<td>‘LOGNORMAL’</td>
</tr>
<tr>
<td>Negative binomial</td>
<td>‘NEGBINOMIAL’</td>
</tr>
<tr>
<td>Normal</td>
<td>‘NORMAL’</td>
</tr>
<tr>
<td>Normal mixture</td>
<td>‘NORMALMIX’</td>
</tr>
<tr>
<td>Pareto</td>
<td>‘PORETO’</td>
</tr>
<tr>
<td>Poisson</td>
<td>‘POISSON’</td>
</tr>
</tbody>
</table>
Distribution Argument

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>'T'</td>
</tr>
<tr>
<td>Uniform</td>
<td>'UNIFORM'</td>
</tr>
<tr>
<td>Wald (inverse Gaussian)</td>
<td>'WALD'</td>
</tr>
<tr>
<td>Weibull</td>
<td>'WEIBULL'</td>
</tr>
</tbody>
</table>

*Note:* Except for T, F, and NORMALMIX, you can minimally identify any distribution by its first four characters.

**quantile**

is a numeric random variable.

**parm-1,...,parm-k**

are optional shape, location, or scale parameters appropriate for the specific distribution.

The LOGSDF function computes the logarithm of the survival function from various continuous and discrete distributions. For more information, see the “PDF Function” on page 695.

---

### LOGSDF Function

**Computes the logarithm of a survival function**

**Category:** Probability

**See:** “SDF Function” on page 819

---

**Syntax**

LOGSDF('dist',quantile,parm-1,...,parm-k)

**Arguments**

**'dist'**

is a character string that identifies the distribution. Valid distributions are as follows:

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bernoulli</td>
<td>'BERNOULLI'</td>
</tr>
<tr>
<td>Beta</td>
<td>'BETA'</td>
</tr>
<tr>
<td>Binomial</td>
<td>'BINOMIAL'</td>
</tr>
<tr>
<td>Cauchy</td>
<td>'CAUCHY'</td>
</tr>
<tr>
<td>Chi-Square</td>
<td>'CHISQUARE'</td>
</tr>
<tr>
<td>Exponential</td>
<td>'EXPONENTIAL'</td>
</tr>
<tr>
<td>Distribution</td>
<td>Argument</td>
</tr>
<tr>
<td>----------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>F</td>
<td>‘F’</td>
</tr>
<tr>
<td>Gamma</td>
<td>‘GAMMA’</td>
</tr>
<tr>
<td>Geometric</td>
<td>‘GEOMETRIC’</td>
</tr>
<tr>
<td>Hypergeometric</td>
<td>‘HYPERGEOMETRIC’</td>
</tr>
<tr>
<td>Laplace</td>
<td>‘LAPLACE’</td>
</tr>
<tr>
<td>Logistic</td>
<td>‘LOGISTIC’</td>
</tr>
<tr>
<td>Lognormal</td>
<td>‘LOGNORMAL’</td>
</tr>
<tr>
<td>Negative binomial</td>
<td>‘NEGBINOMIAL’</td>
</tr>
<tr>
<td>Normal</td>
<td>‘NORMAL’</td>
</tr>
<tr>
<td>Normal mixture</td>
<td>‘NORMALMIX’</td>
</tr>
<tr>
<td>Pareto</td>
<td>‘PARETO’</td>
</tr>
<tr>
<td>Poisson</td>
<td>‘POISSON’</td>
</tr>
<tr>
<td>T</td>
<td>‘T’</td>
</tr>
<tr>
<td>Uniform</td>
<td>‘UNIFORM’</td>
</tr>
<tr>
<td>Wald (inverse Gaussian)</td>
<td>‘WALD’</td>
</tr>
<tr>
<td>Weibull</td>
<td>‘WEIBULL’</td>
</tr>
</tbody>
</table>

**Note:** Except for T, F, and NORMALMIX, you can minimally identify any distribution by its first four characters.

**quantile**

is a numeric random variable.

**parm-1,...,parm-k**

are optional shape, location, or scale parameters appropriate for the specific distribution.

The LOGSDF function computes the logarithm of the survival function from various continuous and discrete distributions. For more information, see the “SDF Function” on page 819.

---

**LOWCASE Function**

**Converts all letters in an argument to lowercase**

**Category:** Character

**Syntax**

LOWCASE(argument)
Arguments

*argument*

specifies any SAS character expression.

Details

If the LOWCASE function returns a value to a variable that has not yet been assigned a length, by default the variable length is determined by the length of the first argument. The LOWCASE function copies a character argument, converts all uppercase letters to lowercase letters, and returns the altered value as a result.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x='INTRODUCTION'; y=lowcase(x); put y;</td>
<td>introduction</td>
</tr>
</tbody>
</table>

See Also

Functions:

“UPCASE Function” on page 867
“PROPCASE Function” on page 737

MAD Function

Returns the median absolute deviation from the median

Category: Descriptive Statistics

Syntax

\[
\text{MAD}(\text{value-1} <, \text{value-2}...>)
\]

Arguments

*value*

specifies a numeric constant, variable, or expression of which the median absolute deviation from the median is to be computed.
Details

If all arguments have missing values, the result is a missing value. Otherwise, the result is the median absolute deviation from the median of the non-missing values. The formula for the median is the same as the one that is used in the UNIVARIATE procedure. For more information, see Base SAS Procedures Guide.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>mad=mad(2,4,1,3,5,999999); put mad;</td>
<td>1.5</td>
</tr>
</tbody>
</table>

See Also

Functions:
- “IQR Function” on page 611
- “MEDIAN Function” on page 645
- “PCTL Function” on page 694

MAX Function

Returns the largest value

Category: Descriptive Statistics

Syntax

MAX(argument,argument,...)

Arguments

argument

is numeric. At least two arguments are required. The argument list may consist of a variable list, which is preceded by OF.

Comparisons

The MAX function returns a missing value (.) only if all arguments are missing.

The MAX operator (<>) returns a missing value only if both operands are missing. In this case, it returns the value of the operand that is higher in the sort order for missing values.
Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=max(8,3);</td>
<td>8</td>
</tr>
<tr>
<td>x1=max(2,6,.);</td>
<td>6</td>
</tr>
<tr>
<td>x2=max(2.-3,1,-1);</td>
<td>2</td>
</tr>
<tr>
<td>x3=max(3,.,-3);</td>
<td>3</td>
</tr>
<tr>
<td>x4=max(of x1-x3);</td>
<td>6</td>
</tr>
</tbody>
</table>

MDY Function

Returns a SAS date value from month, day, and year values

Category: Date and Time

Syntax

MDY(month, day, year)

Arguments

- **month**
  - specifies a numeric expression that represents an integer from 1 through 12.

- **day**
  - specifies a numeric expression that represents an integer from 1 through 31.

- **year**
  - specifies a two-digit or four-digit integer that represents the year. The YEARCUTOFF= system option defines the year value for two-digit dates.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>birthday=mdy(8,27,90);</td>
<td>11196</td>
</tr>
<tr>
<td>put birthday;</td>
<td></td>
</tr>
<tr>
<td>put birthday= worddate.;</td>
<td>birthday=August 27, 1990</td>
</tr>
<tr>
<td>anniversary=mdy(7,11,2001);</td>
<td>15167</td>
</tr>
<tr>
<td>put anniversary;</td>
<td></td>
</tr>
<tr>
<td>put anniversary=date9.;</td>
<td>anniversary=11JUL2001</td>
</tr>
</tbody>
</table>
See Also

Functions:
“DAY Function” on page 481
“MONTH Function” on page 655
“YEAR Function” on page 916

MEAN Function

Returns the arithmetic mean (average)
Category: Descriptive Statistics

Syntax

MEAN(argument<,argument,...>)

Arguments

argument
is numeric. At least one non-missing argument is required. Otherwise, the function returns a missing value.
Tip: The argument list can consist of a variable list, which is preceded by OF.

Comparisons

The GEOMEAN function returns the geometric mean, the HARMEAN function returns the harmonic mean, and the MEDIAN function returns the median of the non-missing values, whereas the MEAN function returns the arithmetic mean (average).

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1=mean(2,.,.,6);</td>
<td>4</td>
</tr>
<tr>
<td>x2=mean(1,2,3,2);</td>
<td>2</td>
</tr>
<tr>
<td>x3=mean(of x1-x2);</td>
<td>3</td>
</tr>
</tbody>
</table>

See Also

Function:
“GEOMEAN Function” on page 566
**MEDIAN Function**

Computes median values

Category: Descriptive Statistics

**Syntax**

\[ \text{MEDIAN}(value_1, value_2, ...) \]

**Arguments**

- \( value \)

  is numeric.

**Details**

The MEDIAN function returns the median of the nonmissing values. If all arguments have missing values, the result is a missing value.

*Note:* The formula that is used in the MEDIAN function is the same as the formula that is used in PROC UNIVARIATE. For more information, see “SAS Elementary Statistics Procedures” in *Base SAS Procedures Guide*.

**Comparisons**

The MEDIAN function returns the median of nonmissing values, whereas the MEAN function returns the arithmetic mean (average).

**Examples**

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = \text{median}(2, 4, 1, 3) );</td>
<td>2.5</td>
</tr>
<tr>
<td>( y = \text{median}(5, 8, 0, 3, 4) );</td>
<td>4</td>
</tr>
</tbody>
</table>
See Also

Function:
“MEAN Function” on page 644

MIN Function

Returns the smallest value
Category: Descriptive Statistics

Syntax
MIN(argument,argument,...)

Arguments

argument is numeric. At least two arguments are required. The argument list may consist of a variable list, which is preceded by OF.

Comparisons
The MIN function returns a missing value (.) only if all arguments are missing. The MIN operator (><) returns a missing value only if either operand is missing. In this case, it returns the value of the operand that is lower in the sort order for missing values.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=min(7,4);</td>
<td>4</td>
</tr>
<tr>
<td>x1=min(2,..,6);</td>
<td>2</td>
</tr>
<tr>
<td>x2=min(2,-3,1,-1);</td>
<td>-3</td>
</tr>
<tr>
<td>x3=min(0,4);</td>
<td>0</td>
</tr>
<tr>
<td>x4=min(of x1-x3);</td>
<td>-3</td>
</tr>
</tbody>
</table>

MINUTE Function

Returns the minute from a SAS time or datetime value
Category: Date and Time

Syntax

\[\text{MINUTE}(\text{time} \mid \text{datetime})\]

Arguments

- \textit{time} specifies a SAS expression that represents a SAS time value.
- \textit{datetime} specifies a SAS expression that represents a SAS datetime value.

Details

The \textsc{minute} function returns an integer that represents a specific minute of the hour. \textsc{minute} always returns a positive number in the range of 0 through 59. Missing values are ignored.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>time='3:19:24't;</td>
<td></td>
</tr>
<tr>
<td>m=minute(time);</td>
<td></td>
</tr>
<tr>
<td>put m;</td>
<td>19</td>
</tr>
</tbody>
</table>

See Also

- Functions:
  - “\textsc{hour} Function” on page 579
  - “\textsc{second} Function” on page 820

MISSING Function

Returns a numeric result that indicates whether the argument contains a missing value

Category: Descriptive Statistics

Category: Character

Syntax

\[\text{MISSING}(\text{numeric-expression} \mid \text{character-expression})\]
Arguments

**numeric-expression**
specifies numeric data.

**character-expression**
is the name of a character variable or an expression that evaluates to a character value.

Details

- The MISSING function checks a numeric or character expression for a missing value, and returns a numeric result. If the argument does not contain a missing value, SAS returns a value of 0. If the argument contains a missing value, SAS returns a value of 1.
- A **character-expression** is defined as having a missing value if the result of the expression contains all blank spaces.
- A **numeric-expression** is defined as having a missing value if the result of the expression is missing (.), or if the expression contains special characters you used to differentiate among missing values. The special characters are the letters A through Z and the underscore, preceded by a period.

Comparisons

The NMISS function requires a numeric argument and returns the number of missing values in the list of arguments.

Examples

This example uses the MISSING function to check whether the input variables contain missing values.

```sas
data values;
  input @1 var1 3. @5 var2 3.;
  if missing(var1) then do;
    put 'Variable 1 is Missing.';
  end;
  else if missing(var2) then do;
    put 'Variable 2 is Missing.';
  end;
datalines;
127
988 195
;
```

In this example, the following message appears in the SAS log.

```
Variable 2 is Missing.
```

See Also

Functions and CALL Routines:
MOD Function

Returns the remainder from the division of the first argument by the second argument, fuzzed to avoid most unexpected floating-point results

Category: Mathematical

Syntax

MOD (argument-1, argument-2)

Arguments

argument-1
is a numeric constant, variable, or expression that specifies the dividend.

argument-2
is a numeric constant, variable, or expression that specifies the divisor.

Restriction: cannot be 0

Details

The MOD function returns the remainder from the division of argument-1 by argument-2. When the result is non-zero, the result has the same sign as the first argument. The sign of the second argument is ignored.

The computation that is performed by the MOD function is exact if both of the following conditions are true:

- Both arguments are exact integers.
- All integers that are less than either argument have exact 8-byte floating-point representations.

To determine the largest integer for which the computation is exact, execute the following DATA step:

```sas
data _null_;  
  exactint = constant('exactint');  
  put exactint=;  
run;
```

Operating Environment Information: You can also refer to the SAS documentation for your operating environment for information about the largest integer.

If either of the above conditions is not true, a small amount of numerical error can occur in the floating-point computation. In this case

- MOD returns zero if the remainder is very close to zero or very close to the value of the second argument.
MOD returns a missing value if the remainder cannot be computed to a precision of approximately three digits or more. In this case, SAS also writes an error message to the log.

Note: Prior to SAS 9, the MOD function did not perform the adjustments to the remainder that were described in the previous paragraph. For this reason, the results of the MOD function in SAS 9 might differ from previous versions.

Comparisons

Here are some comparisons between the MOD and MODZ functions:

- The MOD function performs extra computations, called fuzzing, to return an exact zero when the result would otherwise differ from zero because of numerical error.
- The MODZ function performs no fuzzing.
- Both the MOD and MODZ functions return a missing value if the remainder cannot be computed to a precision of approximately three digits or more.

Examples

The following SAS statements produce results for MOD and MODZ.

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1=mod(10,3);</td>
<td>1.0000</td>
</tr>
<tr>
<td>put x1 9.4;</td>
<td></td>
</tr>
<tr>
<td>xa=modz(10,3);</td>
<td>1.0000</td>
</tr>
<tr>
<td>put xa 9.4;</td>
<td></td>
</tr>
<tr>
<td>x2=mod(.3,-.1);</td>
<td>0.0000</td>
</tr>
<tr>
<td>put x2 9.4;</td>
<td></td>
</tr>
<tr>
<td>xb=modz(.3,-.1);</td>
<td>0.1000</td>
</tr>
<tr>
<td>put xb 9.4;</td>
<td></td>
</tr>
<tr>
<td>x3=mod(1.7,.1);</td>
<td>0.0000</td>
</tr>
<tr>
<td>put x3 9.4;</td>
<td></td>
</tr>
<tr>
<td>xc=modz(1.7,.1);</td>
<td>0.0000</td>
</tr>
<tr>
<td>put xc 9.4;</td>
<td></td>
</tr>
<tr>
<td>x4=mod(.9,.3);</td>
<td>0.0000000000000000000000</td>
</tr>
<tr>
<td>put x4 24.20;</td>
<td></td>
</tr>
<tr>
<td>xd=modz(.9,.3);</td>
<td>0.000000000000000000005551</td>
</tr>
<tr>
<td>put xd 24.20;</td>
<td></td>
</tr>
</tbody>
</table>

See Also

Functions:

“INT Function” on page 597
“INTZ Function” on page 608
“MODZ Function” on page 653
**MODULEC Function**

Calls an external routine and returns a character value

**Category:** External Routines

**See:** “CALL MODULE Routine” on page 348

**Syntax**

`MODULEC(<cntl-string>, module-name<,argument-1, ..., argument-n>)`

**Details**

For details on the MODULEC function, see “CALL MODULE Routine” on page 348.

**See Also**

CALL Routines:

“CALL MODULE Routine” on page 348

“CALL MODULEI Routine” on page 351

Functions:

“MODULEIC Function” on page 651

“MODULEIN Function” on page 652

“MODULEN Function” on page 652

---

**MODULEIC Function**

Calls an external routine and returns a character value (in IML environment only)

**Category:** External Routines

**Restriction:** MODULEIC can only be invoked from within the IML procedure

**See:** “CALL MODULE Routine” on page 348

**Syntax**

`MODULEIC(<cntl-string>,module-name<,argument-1, ..., argument-n>)`

**Details**

For details on the MODULEIC function, see “CALL MODULE Routine” on page 348.
See Also

CALL Routines:
   “CALL MODULE Routine” on page 348
   “CALL MODULEI Routine” on page 351

Functions:
   “MODULEC Function” on page 651
   “MODULEIN Function” on page 652
   “MODULEN Function” on page 652

MODULEIN Function

Calls an external routine and returns a numeric value (in IML environment only)

Category: External Routines
Restriction: MODULEIN can only be invoked from within the IML procedure
See: “CALL MODULE Routine” on page 348

Syntax

MODULEIN(<cntl-string>,module-name<,argument-1, ..., argument-n>)

Details

For details on the MODULEIN function, see “CALL MODULE Routine” on page 348.

See Also

CALL Routines:
   “CALL MODULE Routine” on page 348
   “CALL MODULEI Routine” on page 351

Functions:
   “MODULEC Function” on page 651
   “MODULEIC Function” on page 651
   “MODULEN Function” on page 652

MODULEN Function

Calls an external routine and returns a numeric value
Category: External Routines
See: “CALL MODULE Routine” on page 348

Syntax
MODULEN(<cntl-string,>module-name<,argument-1, ..., argument-n>)

Details
For details on the MODULEN function, see “CALL MODULE Routine” on page 348.

See Also

CALL Routines:
“CALL MODULE Routine” on page 348
“CALL MODULEI Routine” on page 351

Functions:
“MODULEC Function” on page 651
“MODULEIC Function” on page 651
“MODULEIN Function” on page 652

MODZ Function

Returns the remainder from the division of the first argument by the second argument, using zero fuzzing
Category: Mathematical

Syntax
MODZ (argument-1, argument-2)

Arguments

argument-1
is a numeric constant, variable, or expression that specifies the dividend.

argument-2
is a non-zero numeric constant, variable, or expression that specifies the divisor.

Details
The MODZ function returns the remainder from the division of argument-1 by argument-2. When the result is non-zero, the result has the same sign as the first argument. The sign of the second argument is ignored.
The computation that is performed by the MODZ function is exact if both of the following conditions are true:

- Both arguments are exact integers.
- All integers that are less than either argument have exact 8-byte floating-point representation.

To determine the largest integer for which the computation is exact, execute the following DATA step:

```sas
data _null_;  
exactint = constant('exactint');  
put exactint=;  
run;
```

**Operating Environment Information:** You can also refer to the SAS documentation for your operating environment for information about the largest integer.

If either of the above conditions is not true, a small amount of numerical error can occur in the floating-point computation. For example, when you use exact arithmetic and the result is zero, MODZ might return a very small positive value or a value slightly less than the second argument.

**Comparisons**

Here are some comparisons between the MODZ and MOD functions:

- The MODZ function performs no fuzzing.
- The MOD function performs extra computations, called fuzzing, to return an exact zero when the result would otherwise differ from zero because of numerical error.
- Both the MODZ and MOD functions return a missing value if the remainder cannot be computed to a precision of approximately three digits or more.

**Examples**

The following SAS statements produce results for MOD and MODZ.

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1=mod(10,3);</td>
<td>put x1 9.4; 1.0000</td>
</tr>
<tr>
<td>xa=modz(10,3);</td>
<td>put xa 9.4; 1.0000</td>
</tr>
<tr>
<td>x2=mod(.3,-.1);</td>
<td>put x2 9.4; 0.0000</td>
</tr>
<tr>
<td>xb=modz(.3,-.1);</td>
<td>put xb 9.4; 0.1000</td>
</tr>
<tr>
<td>x3=mod(1.7,.1);</td>
<td>put x3 9.4; 0.0000</td>
</tr>
<tr>
<td>xc=modz(1.7,.1);</td>
<td>put xc 9.4; 0.0000</td>
</tr>
<tr>
<td>x4=mod(.9,.3);</td>
<td>put x4 24.20; 0.0000</td>
</tr>
<tr>
<td>xd=modz(.9,.3);</td>
<td>put xd 24.20; 0.0000</td>
</tr>
</tbody>
</table>
MONTH Function

Returns the month from a SAS date value

Category: Date and Time

Syntax

MONTH(date)

Arguments

date

specifies a SAS expression that represents a SAS date value.

Details

The MONTH function returns a numeric value that represents the month from a SAS date value. Numeric values can range from 1 through 12.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>date='25jan94'd; m=month(date); m;</td>
<td>1</td>
</tr>
</tbody>
</table>

See Also

Functions:

“DAY Function” on page 481
“YEAR Function” on page 916
MOPEN Function

Opens a file by directory id and member name, and returns the file identifier or a 0

Category: External Files
See: MOPEN Function in the documentation for your operating environment.

Syntax
MOPEN(directory-id/member-name<open-mode<,record-length<,record-format>>>)

Arguments

directory-id
specifies the identifier that was assigned when the directory was opened, generally by the DOPEN function.

member-name
specifies the member name in the directory.

open-mode
specifies the type of access to the file:
A APPEND mode allows writing new records after the current end of the file.
I INPUT mode allows reading only (default).
O OUTPUT mode defaults to the OPEN mode specified in the operating environment option in the FILENAME statement or function. If no operating environment option is specified, it allows writing new records at the beginning of the file.
S Sequential input mode is used for pipes and other sequential devices such as hardware ports.
U UPDATE mode allows both reading and writing.
W Sequential update mode is used for pipes and other sequential devices such as ports.

Default: I

record-length
specifies a new logical record length for the file. To use the existing record length for the file, specify a length of 0, or do not provide a value here.

record-format
specifies a new record format for the file. To use the existing record format, do not specify a value here. Valid values are:
B specifies that data is to be interpreted as binary data.
D specifies the default record format.
E specifies the record format that you can edit.
F specifies that the file contains fixed-length records.
P specifies that the file contains printer carriage control in operating environment-dependent record format.

V specifies that the file contains variable-length records.

Details

MOPEN returns the identifier for the file, or 0 if the file could not be opened. You can use a file-id that is returned by the MOPEN function as you would use a file-id returned by the FOPEN function.

CAUTION: Use OUTPUT mode with care. Opening an existing file for output may overwrite the current contents of the file without warning.

The member is identified by directory-id and member-name instead of by a fileref. You can also open a directory member by using FILENAME to assign a fileref to the member, followed by a call to FOPEN. However, when you use MOPEN, you do not have to use a separate fileref for each member.

If the file already exists, the output and update modes default to the operating environment option (append or replace) specified with the FILENAME statement or function. For example,

```sas
%let rc=%sysfunc(filename(file,physical-name,,mod));
%let did=%sysfunc(dopen(&file));
%let fid=%sysfunc(mopen(&did,member-name,o,0,d));
%let rc=%sysfunc(fput(&fid,This is a test.));
%let rc=%sysfunc(fwrite(&fid));
%let rc=%sysfunc(fclose(&fid));
```

If ‘file’ already exists, FWRITE appends the new record instead of writing it at the beginning of the file. However, if no operating environment option is specified with the FILENAME function, the output mode implies that the record be replaced.

If the open fails, use SYSMSG to retrieve the message text.

Operating Environment Information: The term directory in this description refers to an aggregate grouping of files that are managed by the operating environment. Different host operating environments identify such groupings with different names, such as directory, subdirectory, MACLIB, or partitioned data set. For details, see the SAS documentation for your operating environment.

Opening a directory member for output or append is not possible in some operating environments.

Examples

This example assigns the fileref MYDIR to a directory. Then it opens the directory, determines the number of members, retrieves the name of the first member, and opens that member. The last three arguments to MOPEN are the defaults. Note that in a macro statement you do not enclose character strings in quotation marks.

```sas
%let filrf=mydir;
%let rc=%sysfunc(filename(filrf,physical-name));
%let did=%sysfunc(dopen(&filrf));
%let frstname=’ ’;
%let memcount=%sysfunc(dnum(&did));
%if (&memcount > 0) %then
  %do;
    %let frstname =
MORT Function

Returns amortization parameters

Category: Financial

Syntax

MORT(a, p, r, n)

Arguments

a
  is numeric, the initial amount.

p
  is numeric, the periodic payment.

r
  is numeric, the periodic interest rate that is expressed as a fraction.

n

See Also

Functions:
  "DCLOSE Function" on page 481
  "DNUM Function" on page 501
  "DOPEN Function" on page 502
  "DREAD Function" on page 506
  "FCLOSE Function" on page 518
  "FILENAME Function" on page 528
  "FOPEN Function" on page 549
  "FPUT Function" on page 556
  "FWRITE Function" on page 563
  "SYMSG Function" on page 848
is an integer, the number of compounding periods.

**Range:** \( n \geq 0 \)

**Details**

The MORT function returns the missing argument in the list of four arguments from an amortization calculation with a fixed interest rate that is compounded each period. The arguments are related by

\[
p = \frac{ar(1+r)^n}{(1+r)^n - 1}
\]

One missing argument must be provided. It is then calculated from the remaining three. No adjustment is made to convert the results to round numbers.

**Examples**

An amount of $50,000 is borrowed for 30 years at an annual interest rate of 10 percent compounded monthly. The monthly payment can be expressed as

```plaintext
payment=mort(50000, . , .10/12,30*12);
```

The value returned is 438.79. The second argument has been set to missing, which indicates that the periodic payment is to be calculated. The 10 percent nominal annual rate has been converted to a monthly rate of 0.10/12. The rate is the fractional (not the percentage) interest rate per compounding period. The 30 years are converted into 360 months.

**N Function**

Returns the number of nonmissing values

**Category:** Descriptive Statistics

**Syntax**

\[ N(\text{argument, argument}, ...) \]

**Arguments**

\[ \text{argument} \]

is numeric. At least one argument is required. The argument list may consist of a variable list, which is preceded by OF.
Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1=n(1,0,,2,5,);</td>
<td>4</td>
</tr>
<tr>
<td>x2=n(1,2);</td>
<td>2</td>
</tr>
<tr>
<td>x3=n(of x1-x2);</td>
<td>2</td>
</tr>
</tbody>
</table>

NETPV Function

Returns the net present value as a fraction

Category: Financial

Syntax

\[ \text{NETPV}(r,freq,c0,c1,\ldots,cn) \]

- \( r \) is numeric, the interest rate over a specified base period of time expressed as a fraction.
- \( freq \) is numeric, the number of payments during the base period of time that is specified with the rate \( r \).
  - Range: \( freq > 0 \)
  - Exception: The case \( freq = 0 \) is a flag to allow continuous discounting.
- \( c0,c1,\ldots,cn \) are numeric cash flows that represent cash outlays (payments) or cash inflows (income) occurring at times 0, 1, \ldots,\( n \). These cash flows are assumed to be equally spaced, beginning-of-period values. Negative values represent payments, positive values represent income, and values of 0 represent no cash flow at a given time. The \( c0 \) argument and the \( c1 \) argument are required.

Details

The NETPV function returns the net present value at time 0 for the set of cash payments \( c0,c1,\ldots,cn \), with a rate \( r \) over a specified base period of time. The argument \( freq > 0 \) describes the number of payments that occur over the specified base period of time.

The net present value is given by

\[
\text{NETPV} (r, freq, c0, c1, \ldots, cn) = \sum_{i=0}^{n} c_i x^i
\]
where

\[ x = \begin{cases} \frac{1}{(1+r)^{1/freq}} & \text{freq} > 0 \\ e^{-r} & \text{freq} = 0 \end{cases} \]

Missing values in the payments are treated as 0 values. When \( freq > 0 \), the rate \( r \) is the effective rate over the specified base period. To compute with a quarterly rate (the base period is three months) of 4 percent with monthly cash payments, set \( freq \) to 3 and set \( r \) to .04.

If \( freq \) is 0, continuous discounting is assumed. The base period is the time interval between two consecutive payments, and the rate \( r \) is a nominal rate.

To compute with a nominal annual interest rate of 11 percent discounted continuously with monthly payments, set \( freq \) to 0 and set \( r \) to \( .11/12 \).

**Examples**

For an initial investment of $500 that returns biannual payments of $200, $300, and $400 over the succeeding 6 years and an annual discount rate of 10 percent, the net present value of the investment can be expressed as

\[ \text{value} = \text{netpv}(0.1, 0.5, -500, 200, 300, 400); \]

The value returned is 95.98.

---

**NLDATE Function**

Converts the SAS date value to the date value of the specified locale using the date-format modifiers

*Category:* Date and Time

*See:* The NLDATE function in *SAS National Language Support (NLS): User's Guide*

---

**NLDATM Function**

Converts the SAS datetime values to the time value of the specified locale using the datetime format modifiers

*Category:* Date and Time

*See:* The NLDATM function in *SAS National Language Support (NLS): User's Guide*

---

**NLTIME Function**

Converts the SAS time or datetime value to the time value of the specified locale using the time-format modifiers
NLITERAL Function

Converts a character string that you specify to a SAS name literal (n-literal)

Category: Character

Syntax

NLITERAL(string)

Arguments

string

specifies a character constant, variable, or expression that is to be converted to a SAS name literal (n-literal).

Tip: Enclose a literal string of characters in quotation marks.

Restriction: If the string is a valid SAS variable name, by default, it is not converted to an n-literal.

Details

If the NLITERAL function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

The NLITERAL function converts string to a SAS name literal (n-literal). String will be converted to an n-literal, unless it qualifies under the default rules for a SAS variable name. These default rules are in effect when the SAS system option VALIDVARNAME=V7:

- It begins with an English letter or an underscore.
- All subsequent characters are English letters, underscores, or digits.
- The length is 32 or fewer alphanumeric characters.

String, therefore, qualifies as a SAS variable name, when all of these are true.

CAUTION:

If insufficient space is available for the resulting n-literal, NLITERAL returns a blank string, prints an error message, and sets _ERROR_ to 1.

The NLITERAL function encloses the value of string in single or double quotation marks, based on the contents of string.
If `string` contains ...

<table>
<thead>
<tr>
<th>string component(s)</th>
<th>NLITERAL encloses the result in ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>an ampersand (&amp;)</td>
<td>single quotation marks</td>
</tr>
<tr>
<td>a percent sign (%)</td>
<td>single quotation marks</td>
</tr>
<tr>
<td>more double quotation marks than single quotation marks</td>
<td>single quotation marks</td>
</tr>
<tr>
<td>none of the above</td>
<td>double quotation marks</td>
</tr>
</tbody>
</table>

### Examples

This example demonstrates multiple uses of NLITERAL.

```plaintext
data test;
  input string $32.;
  length result $ 67;
  result = nliteral(string);
datalines;
abc_123
This and That
cats & dogs
Company’s profits (%)
"Double Quotes"
’Single Quotes’
;
proc print;
title 'Strings Converted to N-Literals or Returned Unchanged';
run;
```

### Output 4.28  Converting Strings to Name Literals with NLITERAL

<table>
<thead>
<tr>
<th>Obs</th>
<th>string</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>abc_123</td>
<td>abc_123</td>
</tr>
<tr>
<td>2</td>
<td>This and That</td>
<td>&quot;This and That&quot;N</td>
</tr>
<tr>
<td>3</td>
<td>cats &amp; dogs</td>
<td>'cats &amp; dogs'N</td>
</tr>
<tr>
<td>4</td>
<td>Company’s profits (%)</td>
<td>'Company’’s profits (%)'N</td>
</tr>
<tr>
<td>5</td>
<td>&quot;Double Quotes&quot;</td>
<td>&quot;&quot;Double Quotes&quot;&quot;N</td>
</tr>
<tr>
<td>6</td>
<td>‘Single Quotes’</td>
<td>&quot;'Single Quotes'&quot;N</td>
</tr>
</tbody>
</table>

### See Also

Functions:

“COMPARE Function” on page 443
“DEQUOTE Function” on page 489
“NVALID Function” on page 688

System Option:

“VALIDVARNAME= System Option” on page 1625
“Rules for Words and Names in the SAS Language” in *SAS Language Reference: Concepts*

---

**NMISS Function**

Returns the number of missing values

**Category:** Descriptive Statistics

**Syntax**

\[
\text{NMISS}(\text{argument}<,...,\text{argument-n}>)
\]

**Arguments**

*argument* is numeric. At least one argument is required. The argument list may consist of a variable list, which is preceded by OF.

**Examples**

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x1=\text{nmiss}(1,0,.,2,5,.));</td>
<td>2</td>
</tr>
<tr>
<td>(x2=\text{nmiss}(1,0));</td>
<td>0</td>
</tr>
<tr>
<td>(x3=\text{nmiss}(\text{of x1-x2}));</td>
<td>0</td>
</tr>
</tbody>
</table>

---

**NORMAL Function**

Returns a random variate from a normal distribution

**Category:** Random Number

**See:** “RANNOR Function” on page 780

**Syntax**

\[
\text{NORMAL}(\text{seed})
\]
Arguments

\( seed \)

is an integer.

Range: \( seed < 2^{31} - 1 \)

Note: If \( seed \leq 0 \), the time of day is used to initialize the seed stream.

---

**NOTALNUM Function**

Searches a character string for a non-alphanumeric character and returns the first position at which it is found

Category: Character

**Syntax**

```
NOTALNUM(string <,start>)
```

**Arguments**

\( string \)

specifies the character expression to search.

\( start \)

is an optional integer that specifies the position at which the search should start and the direction in which to search.

**Details**

The results of the NOTALNUM function depend directly on the translation table that is in effect (see “TRANTAB= System Option” on page 1618) and indirectly on the ENCODING and LOCALE system options.

The NOTALNUM function searches a string for the first occurrence of any character that is not a digit or an uppercase or lowercase letter. If such a character is found, NOTALNUM returns the position in the string of that character. If no such character is found, NOTALNUM returns a value of 0.

If you use only one argument, NOTALNUM begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, \( start \), specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of \( start \) is positive, the search proceeds to the right.
- If the value of \( start \) is negative, the search proceeds to the left.
- If the value of \( start \) is less than the negative length of the string, the search begins at the end of the string.
NOTALPHA returns a value of zero when
- the character that you are searching for is not found
- the value of start is greater than the length of the string
- the value of start = 0.

Comparisons

The NOTALPHA function searches a character string for a non-alphabetic character. The ANYALPHA function searches a character string for an alphanumeric character.

Examples

The following example uses the NOTALPHA function to search a string from left to right for non-alphabetic characters.

```sas
data _null_
   string='Next = Last + 1;'
   j=0;
   do until(j=0);
      j=notalpha(string,j+1);
      if j=0 then put +3 "That’s all";
      else do;
         c=substr(string,j,1);
         put +3 j= c=;
      end;
   end;
run;
```

The following lines are written to the SAS log:

```
j=5  c=
j=6  c==
j=7  c=
j=12 c=
j=13 c=+
j=14 c=
j=16 c=;
That’s all
```

See Also

Function:

“ANYALPHA Function” on page 298
Syntax

NOTALPHA(string <,start>)

Arguments

string is the character constant, variable, or expression to search.

start is an optional integer that specifies the position at which the search should start and the direction in which to search.

Details

The results of the NOTALPHA function depend directly on the translation table that is in effect (see “TRANTAB= System Option” on page 1618) and indirectly on the ENCODING and LOCALE system options.

The NOTALPHA function searches a string for the first occurrence of any character that is not an uppercase or lowercase letter. If such a character is found, NOTALPHA returns the position in the string of that character. If no such character is found, NOTALPHA returns a value of 0.

If you use only one argument, NOTALPHA begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, start, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of start is positive, the search proceeds to the right.
- If the value of start is negative, the search proceeds to the left.
- If the value of start is less than the negative length of the string, the search begins at the end of the string.

NOTALPHA returns a value of zero when

- the character that you are searching for is not found
- the value of start is greater than the length of the string
- the value of start = 0.

Comparisons

The NOTALPHA function searches a character string for a non-alphabetic character. The ANYALPHA function searches a character string for an alphabetic character.

Examples

The following example uses the NOTALPHA function to search a string from left to right for non-alphabetic characters.

```plaintext
data _null_
  string='Next = _n_ + 12E3;'
  j=0;
  do until(j=0);
    j=notalpha(string,j+1);
    if j=0 then put +3 "That's all";
    else do;
      c=substr(string,j,1);
```

put +3 j= c=;
end;
end;
run;

The following lines are written to the SAS log:

j=5 c=
j=6 c==
j=7 c=
j=8 c=_
j=10 c=_
j=11 c=
j=12 c=+
j=13 c=
j=14 c=1
j=15 c=2
j=17 c=3
j=18 c=;
That’s all

See Also

Function:
“ANYALPHA Function” on page 300

---

**NOTCNTRL Function**

Searches a character string for a character that is not a control character and returns the first position at which it is found

**Category:** Character

**Syntax**

`NOTCNTRL(string<,start>)`

**Arguments**

`string`

is the character constant, variable, or expression to search.

`start`

is an optional integer that specifies the position at which the search should start and the direction in which to search.

**Details**

The NOTCNTRL function searches a string for the first occurrence of a character that is not a control character. If such a character is found, NOTCNTRL returns the position
in the string of that character. If no such character is found, NOTCNTRL returns a value of 0.

If you use only one argument, NOTCNTRL begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, \textit{start}, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of \textit{start} is positive, the search proceeds to the right.
- If the value of \textit{start} is negative, the search proceeds to the left.
- If the value of \textit{start} is less than the negative length of the string, the search begins at the end of the string.

NOTCNTRL returns a value of zero when

- the character that you are searching for is not found
- the value of \textit{start} is greater than the length of the string
- the value of \textit{start} = 0.

\textbf{Comparisons}

The NOTCNTRL function searches a character string for a character that is not a control character. The ANYCNTRL function searches a character string for a control character.

\textbf{See Also}

Function:

“ANYCNTRL Function” on page 301

\textbf{NOTDIGIT Function}

Searches a character string for any character that is not a digit and returns the first position at which that character is found

\textbf{Category:} Character

\textbf{Syntax}

\texttt{NOTDIGIT(string \textless;start\textgreater;)}

\textbf{Arguments}

\textit{string} is the character constant, variable, or expression to search.

\textit{start} is an optional integer that specifies the position at which the search should start and the direction in which to search.
Details

The NOTDIGIT function searches a string for the first occurrence of any character that is not a digit. If such a character is found, NOTDIGIT returns the position in the string of that character. If no such character is found, NOTDIGIT returns a value of 0.

If you use only one argument, NOTDIGIT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, \( \text{start} \), specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of \( \text{start} \) is positive, the search proceeds to the right.
- If the value of \( \text{start} \) is negative, the search proceeds to the left.
- If the value of \( \text{start} \) is less than the negative length of the string, the search begins at the end of the string.

NOTDIGIT returns a value of zero when

- the character that you are searching for is not found
- the value of \( \text{start} \) is greater than the length of the string
- the value of \( \text{start} = 0 \).

Comparisons

The NOTDIGIT function searches a character string for any character that is not a digit. The ANYDIGIT function searches a character string for a digit.

Examples

The following example uses the NOTDIGIT function to search for a character that is not a digit.

```sas
data _null_
string='Next = _n_ + 12E3;';
j=0;
do until(j=0);
   j=notdigit(string,j+1);
   if j=0 then put +3 "That's all";
   else do;
      c=substr(string,j,1);
      put +3 j= c=;
   end;
end;
run;
```

The following lines are written to the SAS log:

```
j=1 c=N
j=2 c=e
j=3 c=x
j=4 c=t
j=5 c=
j=6 c==
j=7 c=
j=8 c=_
j=9 c=n
j=10 c=_
j=11 c=
j=12 c=+
```
NOTE Function

Returns an observation ID for the current observation of a SAS data set

Category: SAS File I/O

Syntax

NOTE(data-set-id)

Arguments

data-set-id

specifies the data set identifier that the OPEN function returns.

Details

You can use the observation ID value to return to the current observation by using POINT. Observations can be marked by using NOTE and then returned to later by using POINT. Each observation ID is a unique numeric value.

To free the memory that is associated with an observation ID, use DROPNOTE.

Examples

This example calls CUROBS to display the observation number, calls NOTE to mark the observation, and calls POINT to point to the observation that corresponds to NOTEID.

```sas
%let dsid=%sysfunc(open(sasuser.fitness,i));
/* Go to observation 10 in data set */
%let rc=%sysfunc(fetchobs(&dsid,10));
%if %sysfunc(abs(&rc)) %then
  %put FETCHOBS FAILED;
%else
  %do;
    /* Display observation number */
    /* in the Log */
    %let cur=%sysfunc(curobs(&dsid));
```

See Also

Function:

“ANYDIGIT Function” on page 302
%put CUROBS=&cur;
/* Mark observation 10 using NOTE */
%let noteid=%sysfunc(note(&dsid));
/* Rewind pointer to beginning */
/* of data */
/* set using REWIND */
%let rc=%sysfunc(rewind(&dsid));
/* FETCH first observation into DDV */
%let rc=%sysfunc(fetch(&dsid));
/* Display first observation number */
%let cur=%sysfunc(curobs(&dsid));
%put CUROBS=&cur;
/* POINT to observation 10 marked */
/* earlier by NOTE */
%let rc=%sysfunc(point(&dsid,&noteid));
/* FETCH observation into DDV */
%let rc=%sysfunc(fetch(&dsid));
/* Display observation number 10 */
/* marked by NOTE */
%let cur=%sysfunc(curobs(&dsid));
%put CUROBS=&cur;
%end;
%if (&dsid > 0) %then
  %let rc=%sysfunc(close(&dsid));

The output produced by this program is:

CUROBS=10
CUROBS=1
CUROBS=10

See Also

Functions:
  “DROPNOTE Function” on page 507
  “OPEN Function” on page 690
  “POINT Function” on page 715
  “REWIND Function” on page 787

---

NOTFIRST Function

Searches a character string for an invalid first character in a SAS variable name under VALIDVARNAME=V7, and returns the first position at which that character is found.

Category: Character

Syntax

NOTFIRST(string <,start>)
Arguments

**string**

is the character constant, variable, or expression to search.

**start**

is an optional integer that specifies the position at which the search should start and the direction in which to search.

Details

The NOTFIRST function does not depend on the TRANTAB, ENCODING, or LOCALE options.

The NOTFIRST function searches a string for the first occurrence of any character that is not valid as the first character in a SAS variable name under VALIDVARNAME=V7. These characters are any except the underscore (_) and uppercase or lowercase English letters. If such a character is found, NOTFIRST returns the position in the string of that character. If no such character is found, NOTFIRST returns a value of 0.

If you use only one argument, NOTFIRST begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, **start**, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of **start** is positive, the search proceeds to the right.
- If the value of **start** is negative, the search proceeds to the left.
- If the value of **start** is less than the negative length of the string, the search begins at the end of the string.

NOTFIRST returns a value of zero when

- the character that you are searching for is not found
- the value of **start** is greater than the length of the string
- the value of **start** = 0.

Comparisons

The NOTFIRST function searches a string for the first occurrence of any character that is not valid as the first character in a SAS variable name under VALIDVARNAME=V7. The ANYFIRST function searches a string for the first occurrence of any character that is valid as the first character in a SAS variable name under VALIDVARNAME=V7.

Examples

The following example uses the NOTFIRST function to search a string for any character that is not valid as the first character in a SAS variable name under VALIDVARNAME=V7.

```bash
data _null_
  string='Next = _n_ + 12E3;'
  j=0;
  do until(j=0);
    j=notfirst(string,j+1);
    if j=0 then put +3 "That’s all";
    else do;
      c=substr(string,j,1);
```
put +3 j= c=;
end;
end;
run;

The following lines are written to the SAS log:

j=5 c=
j=6 c==
j=7 c=
j=11 c=
j=12 c=+
j=13 c=
j=14 c=1
j=15 c=2
j=17 c=3
j=18 c=;
That’s all

See Also

Function:
“ANYFIRST Function” on page 304

NOTGRAPH Function

Searches a character string for a non-graphical character and returns the first position at which it is found

Category: Character

Syntax

NOTGRAPH(string <,start>)

Arguments

string
is the character constant, variable, or expression to search.

start
is an optional integer that specifies the position at which the search should start and the direction in which to search.

Details

The results of the NOTGRAPH function depend directly on the translation table that is in effect (see “TRANTAB= System Option” on page 1618) and indirectly on the ENCODING and LOCALE system options.
The NOTGRAPH function searches a string for the first occurrence of a non-graphical character. A graphical character is defined as any printable character other than white space. If such a character is found, NOTGRAPH returns the position in the string of that character. If no such character is found, NOTGRAPH returns a value of 0.

If you use only one argument, NOTGRAPH begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, start, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of start is positive, the search proceeds to the right.
- If the value of start is negative, the search proceeds to the left.
- If the value of start is less than the negative length of the string, the search begins at the end of the string.

NOTGRAPH returns a value of zero when

- the character that you are searching for is not found
- the value of start is greater than the length of the string
- the value of start = 0.

**Comparisons**

The NOTGRAPH function searches a character string for a non-graphical character. The ANYGRAPH function searches a character string for a graphical character.

**Examples**

The following example uses the NOTGRAPH function to search a string for a non-graphical character.

```sas
data _null_;    
  string='Next = _n_ + 12E3;';    
  j=0;    
  do until(j=0);    
    j=notgraph(string,j+1);    
    if j=0 then put +3 "That’s all";    
    else do;    
      c=substr(string,j,1);    
      put +3 j= c=;    
    end;    
  end;    
run;
```

The following lines are written to the SAS log:

```
j=5 c=  
j=7 c=  
j=11 c=  
j=13 c=  
That’s all
```

**See Also**

Function:

“ANYGRAPH Function” on page 306
NOTLOWER Function

Searches a character string for a character that is not a lowercase letter and returns the first position at which that character is found

Category: Character

Syntax

NOTLOWER(string <,start>)

Arguments

string
is the character constant, variable, or expression to search.

start
is an optional integer that specifies the position at which the search should start and the direction in which to search.

Details

The results of the NOTLOWER function depend directly on the translation table that is in effect (see “TRANTAB= System Option” on page 1618) and indirectly on the ENCODING and LOCALE system options.

The NOTLOWER function searches a string for the first occurrence of any character that is not a lowercase letter. If such a character is found, NOTLOWER returns the position in the string of that character. If no such character is found, NOTLOWER returns a value of 0.

If you use only one argument, NOTLOWER begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, start, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of start is positive, the search proceeds to the right.
- If the value of start is negative, the search proceeds to the left.
- If the value of start is less than the negative length of the string, the search begins at the end of the string.

NOTLOWER returns a value of zero when

- the character that you are searching for is not found
- the value of start is greater than the length of the string
- the value of start = 0.

Comparisons

The NOTLOWER function searches a character string for a character that is not a lowercase letter. The ANYLOWER function searches a character string for a lowercase letter.

Examples

The following example uses the NOTLOWER function to search a string for any character that is not a lowercase letter.
data _null_
  string='Next = _n_ + 12E3;'
  j=0;
  do until(j=0);
    j=notlower(string,j+1);
    if j=0 then put +3 "That's all";
    else do;
      c=substr(string,j,1);
      put +3 j= c=;
    end;
  end;
run;

The following lines are written to the SAS log:

j=1 c=N
j=5 c=
j=6 c==
j=7 c=
j=8 c=_
j=10 c=_
j=11 c=_
j=12 c=+
j=13 c=
j=14 c=1
j=15 c=2
j=16 c=E
j=17 c=3
j=18 c=;
That’s all

See Also

Function:
"ANYLOWER Function" on page 307

**NOTNAME Function**

Searches a character string for an invalid character in a SAS variable name under VALIDVARNAME=V7, and returns the first position at which that character is found

Category: Character

**Syntax**

```
NOTNAME(string <,start>)
```
### Arguments

**string**
- is the character constant, variable, or expression to search.

**start**
- is an optional integer that specifies the position at which the search should start and the direction in which to search.

### Details

The NOTNAME function does not depend on the TRANTAB, ENCODING, or LOCALE options.

The NOTNAME function searches a string for the first occurrence of any character that is not valid in a SAS variable name under VALIDVARNAME=V7. These characters are any except underscore (_), digits, and uppercase or lowercase English letters. If such a character is found, NOTNAME returns the position in the string of that character. If no such character is found, NOTNAME returns a value of 0.

If you use only one argument, NOTNAME begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, `start`, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of `start` is positive, the search proceeds to the right.
- If the value of `start` is negative, the search proceeds to the left.
- If the value of `start` is less than the negative length of the string, the search begins at the end of the string.

NOTNAME returns a value of zero when
- the character that you are searching for is not found
- the value of `start` is greater than the length of the string
- the value of `start` = 0.

### Comparisons

The NOTNAME function searches a string for the first occurrence of any character that is not valid in a SAS variable name under VALIDVARNAME=V7. The ANYNAME function searches a string for the first occurrence of any character that is valid in a SAS variable name under VALIDVARNAME=V7.

### Examples

The following example uses the NOTNAME function to search a string for any character that is not valid in a SAS variable name under VALIDVARNAME=V7.

```sas
data _null_
  string='Next = _n_ + 12E3;'
  j=0;
  do until(j=0);
    j=notname(string,j+1);
    if j=0 then put +3 "That’s all";
    else do;
      c=substr(string,j,1);
      put +3 j= c=;
    end;
```
The following lines are written to the SAS log:

```
j=5  c=
j=6  c==
j=7  c=
j=11 c=
j=12 c=+
j=13 c=
j=18 c=;
That's all
```

See Also

Function:
“ANYNAME Function” on page 309

---

**NOTPRINT Function**

Searches a character string for a non-printable character and returns the first position at which it is found

**Category:** Character

**Syntax**

NOTPRINT\(\text{string <,start>}\)

**Arguments**

*string*  
is the character constant, variable, or expression to search.

*start*  
is an optional integer that specifies the position at which the search should start and the direction in which to search.

**Details**

The results of the NOTPRINT function depend directly on the translation table that is in effect (see “TRANTAB= System Option” on page 1618) and indirectly on the ENCODING and LOCALE system options.

The NOTPRINT function searches a string for the first occurrence of a non-printable character. If such a character is found, NOTPRINT returns the position in the string of that character. If no such character is found, NOTPRINT returns a value of 0.

If you use only one argument, NOTPRINT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*,
specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of \textit{start} is positive, the search proceeds to the right.
- If the value of \textit{start} is negative, the search proceeds to the left.
- If the value of \textit{start} is less than the negative length of the string, the search begins at the end of the string.

\texttt{NOTPRINT} returns a value of zero when

- the character that you are searching for is not found
- the value of \textit{start} is greater than the length of the string
- the value of \textit{start} = 0.

\textbf{Comparisons}

The \texttt{NOTPRINT} function searches a character string for a non-printable character. The \texttt{ANYPRINT} function searches a character string for a printable character.

\textbf{See Also}

Function:

“\texttt{ANYPRINT Function}” on page 311

\textbf{NOTPUNCT Function}

\textbf{Searches a character string for a character that is not a punctuation character and returns the first position at which it is found}

\textbf{Category:} Character

\textbf{Syntax}

\texttt{NOTPUNCT}(\textit{string < , start }>)

\textbf{Arguments}

\textit{string} is the character constant, variable, or expression to search.

\textit{start} is an optional integer that specifies the position at which the search should start and the direction in which to search.

\textbf{Details}

The results of the \texttt{NOTPUNCT} function depend directly on the translation table that is in effect (see “\texttt{TRANTAB= System Option}” on page 1618) and indirectly on the \texttt{ENCODING} and \texttt{LOCALE} system options.
The NOTPUNCT function searches a string for the first occurrence of a character that is not a punctuation character. If such a character is found, NOTPUNCT returns the position in the string of that character. If no such character is found, NOTPUNCT returns a value of 0.

If you use only one argument, NOTPUNCT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, \textit{start}, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of \textit{start} is positive, the search proceeds to the right.
- If the value of \textit{start} is negative, the search proceeds to the left.
- If the value of \textit{start} is less than the negative length of the string, the search begins at the end of the string.

NOTPUNCT returns a value of zero when

- the character that you are searching for is not found
- the value of \textit{start} is greater than the length of the string
- the value of \textit{start} = 0.

\textbf{Comparisons}

The NOTPUNCT function searches a character string for a character that is not a punctuation character. The ANYPUNCT function searches a character string for a punctuation character.

\textbf{Examples}

The following example uses the NOTPUNCT function to search a string for characters that are not punctuation characters.

```sas
data _null_
    string='Next = _n_ + 12E3;'
    j=0;
    do until(j=0);
        j=notpunct(string,j+1);
        if j=0 then put +3 "That’s all";
        else do;
            c=substr(string,j,1);
            put +3 j= c=;
        end;
    end;
run;
```

The following lines are written to the SAS log:

```
j=1 c=N
j=2 c=e
j=3 c=x
j=4 c=t
j=5 c=
j=7 c=
j=9 c=n
j=11 c=
j=13 c=
j=14 c=1
j=15 c=2
```
NOTSPACE Function

Searches a character string for a character that is not a white-space character (blank, horizontal and vertical tab, carriage return, line feed, form feed) and returns the first position at which it is found.

Category: Character

Syntax

NOTSPACE(string <,start>)

Arguments

string
is the character constant, variable, or expression to search.

start
is an optional integer that specifies the position at which the search should start and the direction in which to search.

Details

The results of the NOTSPACE function depend directly on the translation table that is in effect (see “TRANTAB= System Option” on page 1618) and indirectly on the ENCODING and LOCALE system options.

The NOTSPACE function searches a string for the first occurrence of a character that is not a blank, horizontal tab, vertical tab, carriage return, line feed, or form feed. If such a character is found, NOTSPACE returns the position in the string of that character. If no such character is found, NOTSPACE returns a value of 0.

If you use only one argument, NOTSPACE begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, start, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of start is positive, the search proceeds to the right.
- If the value of start is negative, the search proceeds to the left.
- If the value of start is less than the negative length of the string, the search begins at the end of the string.

See Also

Function:
“ANYPUNCT Function” on page 312
NOTSPACE returns a value of zero when
- the character that you are searching for is not found
- the value of start is greater than the length of the string
- the value of start = 0.

Comparisons
The NOTSPACE function searches a character string for the first occurrence of a character that is not a blank, horizontal tab, vertical tab, carriage return, line feed, or form feed. The ANYSPACE function searches a character string for the first occurrence of a character that is a blank, horizontal tab, vertical tab, carriage return, line feed, or form feed.

Examples
The following example uses the NOTSPACE function to search a string for a character that is not a white-space character.

```sas
data _null_;    
  string='Next = _n_ + 12E3;';
  j=0;
  do until(j=0);
    j=notspace(string,j+1);
    if j=0 then put +3 "That’s all";
    else do;
      c=substr(string,j,1);
      put +3 j= c=;
    end;
  end;
run;
```

The following lines are written to the SAS log:

```
j=1 c=N
j=2 c=e
j=3 c=x
j=4 c=t
j=6 c==
j=8 c=_
j=9 c=n
j=10 c=_
j=12 c=+
j=14 c=1
j=15 c=2
j=16 c=E
j=17 c=3
j=18 c=;
That’s all
```

See Also
Function:
“ANYSPACE Function” on page 314
NOTUPPER Function

Searches a character string for a character that is not an uppercase letter and returns the first position at which that character is found

Category: Character

Syntax

NOTUPPER(string <,start>)

Arguments

string
  is the character constant, variable, or expression to search.

start
  is an optional integer that specifies the position at which the search should start and the direction in which to search.

Details

The results of the NOTUPPER function depend directly on the translation table that is in effect (see “TRANTAB= System Option” on page 1618) and indirectly on the ENCODING and LOCALE system options.

The NOTUPPER function searches a string for the first occurrence of a character that is not an uppercase letter. If such a character is found, NOTUPPER returns the position in the string of that character. If no such character is found, NOTUPPER returns a value of 0.

If you use only one argument, NOTUPPER begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, start, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of start is positive, the search proceeds to the right.
- If the value of start is negative, the search proceeds to the left.
- If the value of start is less than the negative length of the string, the search begins at the end of the string.

NOTUPPER returns a value of zero when

- the character that you are searching for is not found
- the value of start is greater than the length of the string
- the value of start = 0.

Comparisons

The NOTUPPER function searches a character string for a character that is not an uppercase letter. The ANYUPPER function searches a character string for an uppercase letter.

Examples

The following example uses the NOTUPPER function to search a string for any character that is not an uppercase letter.
data _null_;  
  string='Next = _n_ + 12E3;';  
  j=0;  
  do until(j=0);  
    j=notupper(string,j+1);  
    if j=0 then put +3 "That’s all";  
    else do;  
      c=substr(string,j,1);  
      put +3 j= c=;  
    end;  
  end;  
run;

The following lines are written to the SAS log:
  j=2 c=e  
  j=3 c=x  
  j=4 c=t  
  j=5 c=  
  j=6 c==  
  j=7 c=  
  j=8 c=  
  j=9 c=n  
  j=10 c=  
  j=11 c=  
  j=12 c=+  
  j=13 c=  
  j=14 c=1  
  j=15 c=2  
  j=17 c=3  
  j=18 c=;  
That’s all

See Also

Function:
  “ANYUPPER Function” on page 316

NOTXDIGIT Function

Searches a character string for a character that is not a hexadecimal digit and returns the first position at which that character is found

Category: Character

Syntax

NOTXDIGIT(string <,start>)
Arguments

string
is the character constant, variable, or expression to search.

start
is an optional integer that specifies the position at which the search should start and the direction in which to search.

Details

The NOTXDIGIT function searches a string for the first occurrence of any character that is not a digit or an uppercase or lowercase A, B, C, D, E, or F. If such a character is found, NOTXDIGIT returns the position in the string of that character. If no such character is found, NOTXDIGIT returns a value of 0.

If you use only one argument, NOTXDIGIT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, start, specifies the position at which to begin the search. The direction in which to search is determined in the following way:
- If the value of start is positive, the search proceeds to the right.
- If the value of start is negative, the search proceeds to the left.
- If the value of start is less than the negative length of the string, the search begins at the end of the string.

NOTXDIGIT returns a value of zero when
- the character that you are searching for is not found
- the value of start is greater than the length of the string
- the value of start = 0.

Comparisons

The NOTXDIGIT function searches a character string for a character that is not a hexadecimal digit. The ANYXDIGIT function searches a character string for a character that is a hexadecimal digit.

Examples

The following example uses the NOTXDIGIT function to search a string for a character that is not a hexadecimal digit.

data _null_;  
  string='Next = _n_ + 12E3;';  
  j=0;  
  do until(j=0);  
    j=notxdigit(string,j+1);  
    if j=0 then put +3 "That's all";  
    else do;  
      c=substr(string,j,1);  
      put +3 j= c=;  
    end;  
  end;  
run;

The following lines are written to the SAS log:
Functions and CALL Routines

NPV Function 687

\[
\begin{align*}
\text{j=1} & \quad \text{c=N} \\
\text{j=3} & \quad \text{c=x} \\
\text{j=4} & \quad \text{c=t} \\
\text{j=5} & \quad \text{c=} \\
\text{j=6} & \quad \text{c==} \\
\text{j=7} & \quad \text{c=} \\
\text{j=8} & \quad \text{c=_} \\
\text{j=9} & \quad \text{c=n} \\
\text{j=10} & \quad \text{c=_} \\
\text{j=11} & \quad \text{c=} \\
\text{j=12} & \quad \text{c=}  \\
\text{j=13} & \quad \text{c=} \\
\text{j=18} & \quad \text{c=} ; \\
\text{That’s all}
\end{align*}
\]

See Also

Function:
“ANYXDIGIT Function” on page 317

---

NPV Function

Returns the net present value with the rate expressed as a percentage

Category: Financial

Syntax

\[\text{NPV}(r, \text{freq}, c0, c1, \ldots, cn)\]

Arguments

\(r\)

is numeric, the interest rate over a specified base period of time expressed as a percentage.

\(freq\)

is numeric, the number of payments during the base period of time specified with the rate \(r\).

Range: \(freq > 0\)

Exception: The case \(freq = 0\) is a flag to allow continuous discounting.

\(c0, c1, \ldots, cn\)

are numeric cash flows that represent cash outlays (payments) or cash inflows (income) occurring at times 0, 1, \ldots, n. These cash flows are assumed to be equally spaced, beginning-of-period values. Negative values represent payments, positive values represent income, and values of 0 represent no cash flow at a given time. The \(c0\) argument and the \(c1\) argument are required.
Comparisons
The NPV function is identical to NETPV, except that the r argument is provided as a percentage.

NVALID Function
Checks a character string for validity for use as a SAS variable name in a SAS statement

Category:  Character

Syntax
NVALID(string<,validvarname>)

Arguments

string
specifies a character constant, variable, or expression which will be checked to determine if it can be used as a SAS variable name in a SAS statement.

Note:  Trailing blanks are ignored. △
Tip:  Enclose a literal string of characters in quotation marks.

validvarname
is a character constant, variable, or expression that specifies one of the following values:

V7  determines that string is a valid SAS variable name when all three of the following are true:
   □  It begins with an English letter or an underscore.
   □  All subsequent characters are English letters, underscores, or digits.
   □  The length is 32 or fewer alphanumeric characters.

ANY  determines that string is a valid SAS variable name if it contains 32 or fewer characters of any type, including blanks.

NLITERAL  determines that string is a valid SAS variable name if it is in the form of a SAS name literal (‘name’N) or if it is a valid SAS variable name when VALIDVARNAME=V7.

See:  V7 above in this same list.

Default:  If no value is specified, the NVALID function determines that string is a valid SAS variable name based on the value of the SAS system option VALIDVARNAME=.

Details
The NVALID function checks string to determine if it can be used as a SAS variable name in a SAS statement.
The NVALID function returns a value or 1 or 0.
If this condition exists …                        NVALID returns a value of …

| string can be used as a SAS variable name in a SAS statement | 1 |
| string cannot be used as a SAS variable name in a SAS statement | 0 |

Examples

This example determines the validity of specified strings as SAS variable names. The value that is returned by the NVALID function varies with the validvarname argument. The value of 1 is returned when the string is determined to be a valid SAS variable name under the rules for the specified validvarname argument. Otherwise, the value of 0 is returned.

```sas
options validvarname=v7 ls=64;
data string;
   input string $char40.;
v7=nvalid(string,'v7');
any=nvalid(string,'any');
nliteral=nvalid(string,'nliteral');
default=nvalid(string);
datalines;
Toooooooooooooooooooooonooong

OK
Very_Long_But_Still_OK_for_V7
1st_char_is_a_digit
Embedded blank
!$%&*
"Very Loooong N-Literal with """"N
'No closing quotation mark
;

proc print noobs;
title1 'N LITERAL and Validvarname Arguments Determine';
title2 'Invalid (0) and Valid (1) SAS Variable Names';
run;
```
Output 4.29 Determining the Validity of SAS Variable Names with NLITERAL

<table>
<thead>
<tr>
<th>string</th>
<th>v7</th>
<th>any</th>
<th>nliteral</th>
<th>default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toooooooooooooooooooooooooooooooo Long</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>OK</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Very_Long_But_Still_OK_for_V7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1st_char_is_a_digit</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Embedded blank</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>@**)&amp;* 0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
| "Very Loooong N-Literal with \\
| \"N | 0 | 0 | 1 | 0 |
| ‘No closing quotation mark | 0 | 1 | 0 | 0 |

See Also

Functions:
“COMPARE Function” on page 443
“NLITERAL Function” on page 662

System Option:
“VALIDVARNAME= System Option” on page 1625
“Rules for Words and Names in the SAS Language” in SAS Language Reference: Concepts

OPEN Function

Opens a SAS data set

Category: SAS File I/O

Syntax

OPEN(<data-set-name<,mode>>)

Arguments

data-set-name
specifies the SAS data set to be opened. The name should be of the form

<libref:member-name<data-set-options>|

Default: The default value for data-set-name is _LAST_.
Restriction: If you specify the FIRSTOBS= and OBS= data set, they are ignored. All other data set options are valid.

mode
specifies the type of access to the data set:

I  opens the data set in INPUT mode (default). Values can be read but not modified. ‘I’ uses the strongest access mode available in the engine. That is, if the engine supports random access, OPEN defaults to random access. Otherwise, the file is opened in ‘IN’ mode automatically. Files are opened with sequential access and a system level warning is set.

IN  opens the data set in INPUT mode. Observations are read sequentially, and you are allowed to revisit an observation.

IS  opens the data set in INPUT mode. Observations are read sequentially, but you are not allowed to revisit an observation.

Default:  I

Details

OPEN opens a SAS data set (a SAS data set or a SAS SQL view) and returns a unique numeric data set identifier, which is used in most other data set access functions. OPEN returns 0 if the data set could not be opened.

By default, a SAS data set is opened with a control level of RECORD. For details, see the “CNTLLEV= Data Set Option” on page 13 . An open SAS data set should be closed when it is no longer needed. If you open a data set within a DATA step, it will be closed automatically when the DATA step ends.

OPEN defaults to the strongest access mode available in the engine. That is, if the engine supports random access, OPEN defaults to random access when data sets are opened in INPUT or UPDATE mode. Otherwise, data sets are opened with sequential access, and a system-level warning is set.

Examples

□ This example opens the data set PRICES in the library MASTER using INPUT mode. Note that in a macro statement you do not enclose character strings in quotation marks.

```sas
%let dsid=%sysfunc(open(master.prices,i));
%if (&dsid = 0) %then
   %put %sysfunc(sysmsg());
%else
   %put PRICES data set has been opened;
```

□ This example passes values from macro or DATA step variables to be used on data set options. It opens the data set SASUSER.HOUSES, and uses the WHERE= data set option to apply a permanent WHERE clause. Note that in a macro statement you do not enclose character strings in quotation marks.

```sas
%let choice = style="RANCH";
%let dsid=%sysfunc(open(sasuser.houses
   (where=(&choice)),i));
```

See Also

Function:  “CLOSE Function” on page 436
ORDINAL Function

Returns any specified order statistic

Category: Descriptive Statistics

Syntax

\text{ORDINAL}(\text{count}, \text{argument}, \text{argument}, \ldots)

Arguments

\text{count}

is an integer that is less than or equal to the number of elements in the list of arguments.

\text{argument}

is numeric. At least two arguments are required. An argument can consist of a variable list, preceded by OF.

Details

The ORDINAL function sorts the list and returns the \text{count}^{th} argument in the list.

Examples

\begin{tabular}{|l|l|}
\hline
\text{SAS Statements} & \text{Results} \\
\hline
\text{x1=ordinal(4,1,2,3,-4,5,6,7);} & 3 \\
\hline
\end{tabular}

PATHNAME Function

Returns the physical name of a SAS data library or of an external file, or returns a blank

Category: SAS File I/O

Category: External Files

See: PATHNAME Function in the documentation for your operating environment.

Syntax

\text{PATHNAME}(\text{fileref} \mid \text{libref} <,\text{search-ref}>)

Arguments

**fileref**
specifies the fileref assigned to an external file.

**libref**
specifies the libref assigned to a SAS library.

**search-ref**
specifies whether to search for a fileref or a libref.

- **F** specifies a search for a fileref.
- **L** specifies a search for a libref.

Details

PATHNAME returns the physical name of an external file or SAS library, or blank if **fileref** or **libref** is invalid.

If the name of a fileref is identical to the name of a libref, you can use the **search-ref** argument to choose which reference you want to search. If you specify a value of F, SAS searches for a fileref. If you specify a value of L, SAS searches for a libref.

If you do not specify a **search-ref** argument, and the name of a fileref is identical to the name of a libref, PATHNAME searches first for a fileref. If a fileref does not exist, PATHNAME then searches for a libref.

The default length of the target variable in the DATA step is 200 characters.

You can assign a fileref to an external file by using the **FILENAME** statement or the **FILENAME** function.

You can assign a libref to a SAS library using the **LIBNAME** statement or the **LIBNAME** function. Some operating environments allow you to assign a libref using system commands.

Operating Environment Information: Under some operating environments, filerefs can also be assigned by using system commands. For details, see the SAS documentation for your operating environment.

Examples

This example uses the **FILEREF** function to verify that the fileref **MYFILE** is associated with an external file. Then it uses PATHNAME to retrieve the actual name of the external file:

```bash
data _null_
  length fname $ 100;
  rc=fileref("myfile");
  if (rc=0) then do;
    fname=pathname("myfile");
    put fname=;
  end;
run;
```
See Also

Functions:

“FEXIST Function” on page 525
“FILEEXIST Function” on page 527
“FILENAME Function” on page 528
“FILEREF Function” on page 531

Statements:

“LIBNAME Statement” on page 1286
“FILENAME Statement” on page 1172

PCTL Function

Computes percentiles

Category: Descriptive Statistics

Syntax

PCTL\(<n\>(percentage, value1<, value2, ...>)

Arguments

\(n\)

is a digit from 1 to 5 which specifies the definition of the percentile to be computed.

Default: definition 5

\(percentage\)

specifies the percentile to be computed.

Requirement: is numeric where, \(0 \leq percentage \leq 100\).

\(value\)

is numeric.

Details

The PCTL function returns the percentile of the nonmissing values corresponding to the percentage. If \(percentage\) is missing, less than zero, or greater than 100, the PCTL function generates an error message.

Note: The formula that is used in the PCTL function is the same as the function that is used in PROC UNIVARIATE. For more information, see “SAS Elementary Statistics Procedures” in Base SAS Procedures Guide.
Examples

SAS Statements | Results
--- | ---
lower_quartile=PCTL(25,2,4,1,3); put lower_quartile; | 1.5
percentile_def2=PCTL2(25,2,4,1,3); put percentile_def2; | 1
lower_tertile=PCTL(100/3,2,4,1,3); put lower_tertile; | 2
percentile_def3=PCTL3(100/3,2,4,1,3); put percentile_def3; | 2
median=PCTL(50,2,4,1,3); put median; | 2.5
upper_tertile=PCTL(200/3,2,4,1,3); put upper_tertile; | 3
upper_quartile=PCTL(75,2,4,1,3); put upper_quartile; | 3.5

PDF Function

Computes probability density (mass) functions

Category: Probability
Alias: PMF

Syntax

PDF('dist',quantile,parm-1, ...,parm-k)

Arguments

'dist'
is a character string that identifies the distribution. Valid distributions are as follows:

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bernoulli</td>
<td>'BERNOULLI'</td>
</tr>
<tr>
<td>Beta</td>
<td>'BETA'</td>
</tr>
<tr>
<td>Binomial</td>
<td>'BINOMIAL'</td>
</tr>
<tr>
<td>Cauchy</td>
<td>'CAUCHY'</td>
</tr>
<tr>
<td>Chi-Square</td>
<td>'CHISQUARE'</td>
</tr>
</tbody>
</table>
### Distribution Argument

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exponential</td>
<td>‘EXONENTIAL’</td>
</tr>
<tr>
<td>F</td>
<td>‘F’</td>
</tr>
<tr>
<td>Gamma</td>
<td>‘GAMMA’</td>
</tr>
<tr>
<td>Geometric</td>
<td>‘GEOMETRIC’</td>
</tr>
<tr>
<td>Hypergeometric</td>
<td>‘HYPERGEOMETRIC’</td>
</tr>
<tr>
<td>Laplace</td>
<td>‘LAPLACE’</td>
</tr>
<tr>
<td>Logistic</td>
<td>‘LOGISTIC’</td>
</tr>
<tr>
<td>Lognormal</td>
<td>‘LOGNORMAL’</td>
</tr>
<tr>
<td>Negative binomial</td>
<td>‘NEGBINOMIAL’</td>
</tr>
<tr>
<td>Normal</td>
<td>‘NORMAL’</td>
</tr>
<tr>
<td>Normal mixture</td>
<td>‘NORMALMIX’</td>
</tr>
<tr>
<td>Pareto</td>
<td>‘PARETO’</td>
</tr>
<tr>
<td>Poisson</td>
<td>‘POISSON’</td>
</tr>
<tr>
<td>T</td>
<td>‘T’</td>
</tr>
<tr>
<td>Uniform</td>
<td>‘UNIFORM’</td>
</tr>
<tr>
<td>Wald (inverse Gaussian)</td>
<td>‘WALD’</td>
</tr>
<tr>
<td>Weibull</td>
<td>‘WEIBULL’</td>
</tr>
</tbody>
</table>

**Note:** Except for T, F, and NORMALMIX, you can minimally identify any distribution by its first four characters.

Quantile

is a numeric random variable.

**parm-1, ..., parm-k**

are optional shape, location, or scale parameters that are appropriate for the specific distribution.

**See:** “Details” on page 696 for complete information about these parameters

### Details

#### Bernoulli Distribution

**PDF(‘BERNOULLI’, x, p)**

where

- **x**
  - is a numeric random variable.

- **p**
  - is a numeric probability of success.

**Range:** 0 ≤ p ≤ 1

The PDF function for the Bernoulli distribution returns the probability density function of a Bernoulli distribution, with probability of success equal to p. The PDF function is evaluated at the value x. The equation follows:
PDF (\text{iBERN}^t, x, p) = \begin{cases} 
  0 & x < 0 \\
  1 - p & x = 0 \\
  0 & 0 < x < 1 \\
  p & x = 1 \\
  0 & x > 1 
\end{cases}

\textbf{Beta Distribution}

\text{PDF('BETA', x, a, b, l, r)}

where

- \(x\) is a numeric random variable.
- \(a\) is a numeric shape parameter.  
  \textbf{Range:} \(a > 0\)
- \(b\) is a numeric shape parameter.  
  \textbf{Range:} \(b > 0\)
- \(l\) is the numeric left location parameter.  
  \textbf{Default:} 0
- \(r\) is the right location parameter.  
  \textbf{Default:} 0
  \textbf{Range:} \(r > l\)

The PDF function for the beta distribution returns the probability density function of a beta distribution, with shape parameters \(a\) and \(b\). The PDF function is evaluated at the value \(x\). The equation follows:

\[
PDF \left( \text{iBETA}^t, x, a, b, l, r \right) = \begin{cases} 
0 & x < 1 \\
\frac{1}{\beta(a,b)} \frac{(x-l)^{a-1}(x-r)^{b-1}}{(r-l)^{a+b-1}} & l \leq x \leq r \\
0 & x > r 
\end{cases}
\]

\textbf{Binomial Distribution}

\text{PDF('BINOMIAL', m, p, n)}

where

- \(m\) is an integer random variable that counts the number of successes.  
  \textbf{Range:} \(m = 0, 1, \ldots\)
- \(p\) is a numeric probability of success.  
  \textbf{Range:} \(0 \leq p \leq 1\)

\textbf{Note:} There are no location or scale parameters for this distribution. \(\triangle\)
n is an integer parameter that counts the number of independent Bernoulli trials.

**Range:** \( n = 0, 1, ... \)

The PDF function for the binomial distribution returns the probability density function of a binomial distribution, with parameters \( p \) and \( n \), which is evaluated at the value \( m \). The equation follows:

\[
PDF(BINOMIAL, m, p, n) = \begin{cases} 
0 & m < 0 \\
\binom{n}{m} p^m (1 - p)^{n-m} & 0 \leq m \leq n \\
0 & m > n 
\end{cases}
\]

**Note:** There are no location or scale parameters for the binomial distribution.

**Cauchy Distribution**

\[
PDF(CAUCHY, x, \theta, \lambda) 
\]

where

- \( x \) is a numeric random variable.
- \( \theta \) is a numeric location parameter.
- **Default:** 0
- \( \lambda \) is a numeric scale parameter.
- **Default:** 1
- **Range:** \( \lambda > 0 \)

The PDF function for the Cauchy distribution returns the probability density function of a Cauchy distribution, with the location parameter \( \theta \) and the scale parameter \( \lambda \). The PDF function is evaluated at the value \( x \). The equation follows:

\[
PDF(CAUCHY, x, \theta, \lambda) = \frac{1}{\pi} \left( \frac{\lambda}{\lambda^2 + (x - \theta)^2} \right)
\]

**Chi-Square Distribution**

\[
PDF(CHISQUARE, x, df, nc) 
\]

where

- \( x \) is a numeric random variable.
- \( df \) is a numeric degrees of freedom parameter.
- **Range:** \( df > 0 \)
- \( nc \) is an optional numeric non-centrality parameter.
- **Range:** \( nc \geq 0 \)
The PDF function for the chi-square distribution returns the probability density function of a chi-square distribution, with \( df \) degrees of freedom and non-centrality parameter \( nc \). The PDF function is evaluated at the value \( x \). This function accepts non-integer degrees of freedom. If \( nc \) is omitted or equal to zero, the value returned is from the central chi-square distribution. The following equation describes the PDF function of the chi-square distribution,

\[
PDF('\text{CHISQ}', x, v, \lambda) = \begin{cases} 
0 & x < 0 \\
\sum_{j=0}^{\infty} e^{-\frac{x}{2}} \frac{x^j}{j!} p_c (x, v + 2j) & x \geq 0
\end{cases}
\]

where \( p_c(.,.) \) denotes the density from the central chi-square distribution:

\[
p_c (x, a) = \frac{1}{2} p_g \left( \frac{x}{2}, \frac{a}{2} \right)
\]

and where \( p_g(y,b) \) is the density from the gamma distribution, which is given by

\[
p_g (y, b) = \frac{1}{\Gamma(b)} e^{-y} y^{b-1}
\]

**Exponential Distribution**

PDF('EXPONENTIAL', x <, \lambda>)

where

- \( x \) is a numeric random variable.
- \( \lambda \) is a scale parameter.

**Default:** 1  
Range: \( \lambda > 0 \)

The PDF function for the exponential distribution returns the probability density function of an exponential distribution, with the scale parameter \( \lambda \). The PDF function is evaluated at the value \( x \). The equation follows:

\[
PDF('EXPO', x, \lambda) = \begin{cases} 
0 & x < 0 \\
\frac{1}{\lambda} exp \left( -\frac{x}{\lambda} \right) & x \geq 0
\end{cases}
\]

**F Distribution**

PDF('F', x, ndf, ddf<, nc>)

where

- \( x \) is a numeric random variable.
- \( ndf \) is a numeric numerator degrees of freedom parameter.
- \( Range: \ ndf > 0 \)
$ddf$

is a numeric denominator degrees of freedom parameter.

**Range:** $ddf > 0$

$nc$

is a numeric non-centrality parameter.

**Range:** $nc \geq 0$

The PDF function for the $F$ distribution returns the probability density function of an $F$ distribution, with $ndf$ numerator degrees of freedom, $ddf$ denominator degrees of freedom, and non-centrality parameter $nc$. The PDF function is evaluated at the value $x$. This PDF function accepts non-integer degrees of freedom for $ndf$ and $ddf$. If $nc$ is omitted or equal to zero, the value returned is from a central $F$ distribution. The following equation describes the PDF function of the $F$ distribution,

$$
PDF(F', x, v_1, v_2, \lambda) = \begin{cases} 
0 & x < 0 \\
gamma \sum_{j=0}^{\infty} e^{-\frac{x}{2}} \left( \frac{x}{2} \right)^j j! - p_f(f, v_1 + 2j, v_2) & x \geq 0 
\end{cases}
$$

where $p_f(f, u_1, u_2)$ is the density from the central $F$ distribution with

$$
p_f(f, u_1, u_2) = p_B \left( \frac{u_1 f}{u_f + u_2}, \frac{u_1}{2}, \frac{u_2}{2} \right) \frac{u_1 u_2}{(u_f + u_1 f)^2}
$$

and where $p_B(x, a, b)$ is the density from the standard beta distribution.

**Note:** There are no location or scale parameters for the $F$ distribution.

**Gamma Distribution**

PDF('GAMMA', $x$, $a$, $<$, $\lambda>$)

where

$x$

is a numeric random variable.

$a$

is a numeric shape parameter.

**Range:** $a > 0$

$\lambda$

is a numeric scale parameter.

**Default:** 1

**Range:** $\lambda > 0$

The PDF function for the gamma distribution returns the probability density function of a gamma distribution, with the shape parameter $a$ and the scale parameter $\lambda$. The PDF function is evaluated at the value $x$. The equation follows:

$$
PDF(Gamma', x, a, \lambda) = \begin{cases} 
\frac{1}{\lambda^a a!} x^{a-1} e^{-x/\lambda} & x < 0 \\
\frac{1}{\lambda^a a!} x^{a-1} e^{-x/\lambda} & x \geq 0 
\end{cases}
$$

**Geometric Distribution**

PDF('GEOMETRIC', $m$, $p$)
where

\( m \)

is a numeric random variable that denotes the number of failures.

**Range:** \( m \geq 0 \)

\( p \)

is a numeric probability of success.

**Range:** \( 0 \leq p \leq 1 \)

The PDF function for the geometric distribution returns the probability density function of a geometric distribution, with parameter \( p \). The PDF function is evaluated at the value \( m \). The equation follows:

\[
PDF(GEOM, m, p) = \begin{cases} 
0 & m < 0 \\
p(1-p)^m & m \geq 0 
\end{cases}
\]

**Note:** There are no location or scale parameters for this distribution.

**Hypergeometric Distribution**

\( CDF(HYPER, x, N, R, n, o) \)

where

\( x \)

is an integer random variable.

\( N \)

is an integer population size parameter.

**Range:** \( N = 1, 2, ... \)

\( R \)

is an integer number of items in the category of interest.

**Range:** \( R = 0, 1, ..., N \)

\( n \)

is an integer sample size parameter.

**Range:** \( n = 1, 2, ..., N \)

\( o \)

is an optional numeric odds ratio parameter.

**Range:** \( o > 0 \)

The PDF function for the hypergeometric distribution returns the probability density function of an extended hypergeometric distribution, with population size \( N \), number of items \( R \), sample size \( n \), and odds ratio \( o \). The PDF function is evaluated at the value \( x \). If \( o \) is omitted or equal to 1, the value returned is from the usual hypergeometric distribution. The equation follows:

\[
PDF(HYPER, x, N, R, n, o) = \begin{cases} 
0 & x < max(0, R + n - N) \\
\frac{\binom{R}{x} \binom{N-R}{n-x} o^x}{\sum_{j=x}^{\min(R,n)} \binom{R}{j} \binom{N-R}{n-j} o^j} & max(0, R + n - N) \leq x \leq \min(R,n) \\
0 & x > \min(R,n) \end{cases}
\]
Laplace Distribution

PDF('LAPLACE', x<θ, λ >)

where

\( x \)

is a numeric random variable.

\( \theta \)

is a numeric location parameter.

Default: 0

\( \lambda \)

is a numeric scale parameter.

Default: 1

Range: \( \lambda > 0 \)

The PDF function for the Laplace distribution returns the probability density function of the Laplace distribution, with the location parameter \( \theta \) and the scale parameter \( \lambda \). The PDF function is evaluated at the value \( x \). The equation follows:

\[
PDF\left('LAPLACE', x, \theta, \lambda \right) = \frac{1}{2\lambda} exp \left( \frac{|x - \theta|}{\lambda} \right)
\]

Logistic Distribution

PDF('LOGISTIC', x<θ, λ >)

where

\( x \)

is a numeric random variable.

\( \theta \)

is a numeric location parameter.

Default: 0

\( \lambda \)

is a numeric scale parameter.

Default: 1

Range: \( \lambda > 0 \)

The PDF function for the logistic distribution returns the probability density function of a logistic distribution, with the location parameter \( \theta \) and the scale parameter \( \lambda \). The PDF function is evaluated at the value \( x \). The equation follows:

\[
PDF\left('LOGISTIC', x, \theta, \lambda \right) = \frac{1}{\lambda \left(1 + exp \left(\frac{x-\theta}{\lambda}\right)\right)^2}
\]

Lognormal Distribution

PDF('LOGNORMAL', x<θ, λ >)

where

\( x \)

is a numeric random variable.
\( \theta \)

is a numeric location parameter.

**Default:** 0

\( \lambda \)

is a numeric scale parameter.

**Default:** 1

**Range:** \( \lambda > 0 \)

The PDF function for the lognormal distribution returns the probability density function of a lognormal distribution, with the location parameter \( \theta \) and the scale parameter \( \lambda \). The PDF function is evaluated at the value \( x \). The equation follows:

\[
PDF \left( ^t LOGN', x, \theta, \lambda \right) = \begin{cases} 
0 & x \leq 0 \\
\frac{1}{x\lambda^{\frac{1}{2}}\pi} \exp \left( -\frac{(\log(x) - \theta)^2}{2\lambda^2} \right) & x > 0
\end{cases}
\]

**Negative Binomial Distribution**

\( PDF(\text{NEGBINOMIAL}', m, p, n) \)

where

\( m \)

is a positive integer random variable that counts the number of failures.

**Range:** \( m = 0, 1, \ldots \)

\( p \)

is a numeric probability of success.

**Range:** \( 0 \leq p \leq 1 \)

\( n \)

is an integer parameter that counts the number of successes.

**Range:** \( n = 1, 2, \ldots \)

The PDF function for the negative binomial distribution returns the probability density function of a negative binomial distribution, with probability of success \( p \) and number of successes \( n \). The PDF function is evaluated at the value \( m \). The equation follows:

\[
PDF \left( ^t NEGB', m, p, n \right) = \begin{cases} 
0 & m < 0 \\
p^n \left( \frac{n + m - 1}{m} \right) (1 - p)^m & m \geq 0
\end{cases}
\]

**Note:** There are no location or scale parameters for the negative binomial distribution.

**Normal Distribution**

\( PDF(\text{NORMAL}', x<\beta, \lambda>) \)

where

\( x \)

is a numeric random variable.

\( \theta \)

is a numeric location parameter.
Default: 0

\( \lambda \) is a numeric scale parameter.

Default: 1

Range: \( \lambda > 0 \)

The PDF function for the normal distribution returns the probability density function of a normal distribution, with the location parameter \( \theta \) and the scale parameter \( \lambda \). The PDF function is evaluated at the value \( x \). The equation follows:

\[
PDF(NORMAL', x, \theta, \lambda) = \frac{1}{\lambda \sqrt{2\pi}} \exp \left( -\frac{(x - \theta)^2}{2\lambda^2} \right)
\]

Normal Mixture Distribution

CDF('NORMALMIX', x, n, p, m, s)

where

- \( x \) is a numeric random variable.
- \( n \) is the integer number of mixtures.
  
_range: \( n = 1, 2, ... \)
- \( p \)
  
_is the \( n \) proportions, \( p_1, p_2, \ldots, p_n \), where \( \sum_{i=1}^{i=n} p_i = 1. \)
- **Range:** \( p = 0, 1, ... \)
- \( m \)
  
_is the \( n \) means \( m_1, m_2, \ldots, m_n \).
- \( s \)
  
_is the \( n \) standard deviations \( s_1, s_2, \ldots, s_n \).
- **Range:** \( s > 0 \)

The PDF function for the normal mixture distribution returns the probability that an observation from a mixture of normal distribution is less than or equal to \( x \). The equation follows:

\[
PDF(NORMALMIX', x, n, p, m, s) = \sum_{i=1}^{i=n} p_i \ PDF(NORMAL', x, m_i, s_i)
\]

**Note:** There are no location or scale parameters for the normal mixture distribution. \( \triangle \)

Pareto Distribution

PDF('PARETO', x, a, k>)

where

- \( x \)
is a numeric random variable.

\( a \)
is a numeric shape parameter.

**Range:** \( a > 0 \)

\( k \)
is a numeric scale parameter.

**Default:** 1

**Range:** \( k > 0 \)

The PDF function for the Pareto distribution returns the probability density function of a Pareto distribution, with the shape parameter \( a \) and the scale parameter \( k \). The PDF function is evaluated at the value \( x \). The equation follows:

\[
PDF ('PARETO', x, a, k) = \begin{cases} 
0 & x < k \\
\frac{a}{k} \left( \frac{k}{x} \right)^{a+1} & x \geq k
\end{cases}
\]

**Poisson Distribution**

\( PDF('POISSON', n, m) \)

where

\( n \)
is an integer random variable.

**Range:** \( n = 0, 1, ... \)

\( m \)
is a numeric mean parameter.

**Range:** \( m > 0 \)

The PDF function for the Poisson distribution returns the probability density function of a Poisson distribution, with mean \( m \). The PDF function is evaluated at the value \( n \). The equation follows:

\[
PDF ('POISSON', n, m) = \begin{cases} 
0 & n < 0 \\
\frac{e^{-m} m^n}{n!} & n \geq 0
\end{cases}
\]

**Note:** There are no location or scale parameters for the Poisson distribution.

**T Distribution**

\( PDF('T', t, df,<,nc>) \)

where

\( t \)
is a numeric random variable.

\( df \)
is a numeric degrees of freedom parameter.

**Range:** \( df > 0 \)

\( nc \)
is an optional numeric non-centrality parameter.

The PDF function for the \( T \) distribution returns the probability density function of a \( T \) distribution, with degrees of freedom \( df \) and non-centrality parameter \( nc \). The PDF
function is evaluated at the value $x$. This PDF function accepts non-integer degrees of freedom. If $nc$ is omitted or equal to zero, the value returned is from the central $T$ distribution. The equation follows:

$$
PDF\left(^tT^', t, v, \delta\right) = \frac{1}{2^{(\frac{v}{2}-1)}\Gamma\left(\frac{v}{2}\right)} \int_0^\infty \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \left(\frac{t^2}{v} - \delta^2\right)^\frac{v}{2} x dx
$$

Note: There are no location or scale parameters for the $T$ distribution. △

**Uniform Distribution**

PDF('UNIFORM', $x$, $l$, $r$)

where

$x$

is a numeric random variable.

$l$

is the numeric left location parameter.

Default: 0

$r$

is the numeric right location parameter.

Default: 1

Range: $r > l$

The PDF function for the uniform distribution returns the probability density function of a uniform distribution, with the left location parameter $l$ and the right location parameter $r$. The PDF function is evaluated at the value $x$. The equation follows:

$$
PDF\left(^t\text{UNIFORM}', x, l, r\right) = \begin{cases} 
0 & x < l \\
\frac{1}{r-l} & l \leq x \leq r \\
0 & x > r
\end{cases}
$$

**Wald (Inverse Gaussian) Distribution**

PDF('WALD', $x$, $d$)

PDF('IGAUSS', $x$, $d$)

where

$x$

is a numeric random variable.

d

is a numeric shape parameter.

Range: $d > 0$

The PDF function for the Wald distribution returns the probability density function of a Wald distribution, with shape parameter $d$, which is evaluated at the value $x$. The equation follows:
PDF \((tWALD', x, d)\) = \[
\begin{cases} 
0 & x \leq 0 \\
\sqrt{\frac{d}{2\pi x^3}} e^{x} \left(-\frac{d}{2}x + d - \frac{d}{2x^2}\right) & x > 0
\end{cases}
\]

Note: There are no location or scale parameters for the Wald distribution.

**Weibull Distribution**

PDF(‘WEIBULL’,x,a<,λ>)

where

\(x\)

is a numeric random variable.

\(a\)

is a numeric shape parameter.

**Range:** \(a > 0\)

\(λ\)

is a numeric scale parameter.

**Default:** 1

**Range:** \(λ > 0\)

The PDF function for the Weibull distribution returns the probability density function of a Weibull distribution, with the shape parameter \(a\) and the scale parameter \(λ\). The PDF function is evaluated at the value \(x\). The equation follows:

\[PDF(‘WEIBULL’, x, a, λ) = \begin{cases} 
0 & x < 0 \\
\exp \left(-\left(\frac{x}{λ}\right)^a\right) \frac{a}{λ} \left(\frac{x}{λ}\right)^{a-1} & x \geq 0
\end{cases}\]

**Examples**

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>y=pdf(‘BERN’,0,.25);</td>
<td>0.75</td>
</tr>
<tr>
<td>y=pdf(‘BERN’,1,.25);</td>
<td>0.25</td>
</tr>
<tr>
<td>y=pdf(‘BETA’,0.2,3,4);</td>
<td>1.2288</td>
</tr>
<tr>
<td>y=pdf(‘BINOM’,4,.5,10);</td>
<td>0.20508</td>
</tr>
<tr>
<td>y=pdf(‘CAUCHY’,2);</td>
<td>0.063662</td>
</tr>
<tr>
<td>y=pdf(‘CHISQ’,11.264,11);</td>
<td>0.081686</td>
</tr>
<tr>
<td>y=pdf(‘EXPO’,1);</td>
<td>0.36788</td>
</tr>
<tr>
<td>y=pdf(‘F’,3,32,2,3);</td>
<td>0.054027</td>
</tr>
<tr>
<td>y=pdf(‘GAMMA’,1,3);</td>
<td>0.18394</td>
</tr>
<tr>
<td>y=pdf(‘HYPER’,2,200,50,10);</td>
<td>0.28685</td>
</tr>
<tr>
<td>y=pdf(‘LAPLACE’,1);</td>
<td>0.18394</td>
</tr>
<tr>
<td>y=pdf(‘LOGISTIC’,1);</td>
<td>0.19661</td>
</tr>
<tr>
<td>y=pdf(‘LOGNORMAL’,1);</td>
<td>0.39894</td>
</tr>
<tr>
<td>y=pdf(‘NEGB’,1,.5,2);</td>
<td>0.25</td>
</tr>
</tbody>
</table>
PEEK Function

Stores the contents of a memory address into a numeric variable on a 32–bit platform

Category: Special
Restriction: Use on 32–bit platforms only.

Syntax

PEEK(address<,length>)

Arguments

address
specifies the memory address.

length
specifies the data length.

Default: a 4-byte address pointer
Range: 2 to 8

Details

If you do not have access to the memory storage location that you are requesting, the PEEK function returns an "Invalid argument" error.

You cannot use the PEEK function on 64-bit platforms. If you attempt to use it, SAS writes a message to the log stating that this restriction applies. If you have legacy applications that use PEEK, change the applications and use PEEKLONG instead. You can use PEEKLONG on both 32–bit and 64–bit platforms.

Comparisons

The PEEK function stores the contents of a memory address into a numeric variable. The PEEKC function stores the contents of a memory address into a character variable.
Note: SAS recommends that you use PEEKLONG instead of PEEK because PEEKLONG can be used on both 32-bit and 64-bit platforms.

Examples

The following example, specific to the z/OS operating environment, returns a numeric value that represents the address of the Communication Vector Table (CVT).

```sas
data _null_; /* 16 is the location of the CVT address */
y=16;
x=peek(y);
put 'x= ' x hex8.;
run;
```

See Also

Functions:
- “ADDR Function” on page 295
- “PEEKC Function” on page 709

CALL Routine:
- “CALL POKE Routine” on page 352

PEEKC Function

Stores the contents of a memory address in a character variable on a 32-bit platform

Category: Special

Restriction: Use on 32-bit platforms only.

Syntax

PEEKC(address<,length>)

Arguments

address
- specifies the memory address.

length
- specifies the data length.

Default: 8, unless the variable length has already been set (by the LENGTH statement, for example)

Range: 1 to 32,767
**Details**

If you do not have access to the memory storage location that you are requesting, the PEEKC function returns an "Invalid argument" error.

You cannot use the PEEK function on 64-bit platforms. If you attempt to use it, SAS writes a message to the log stating that this restriction applies. If you have legacy applications that use PEEKC, change the applications and use PEEKCLONG instead. You can use PEEKCLONG on both 32-bit and 64-bit platforms.

**Comparisons**

The PEEKC function stores the contents of a memory address into a character variable. The PEEK function stores the contents of a memory address into a numeric variable.

*Note:* SAS recommends that you use PEEKCLONG instead of PEEKC because PEEKCLONG can be used on both 32-bit and 64-bit platforms.

**Examples**

**Example 1: Listing ASCB Bytes**  
The following example, specific to the z/OS operating environment, uses both PEEK and PEEKC, and prints the first four bytes of the Address Space Control Block (ASCB).

```sas
data _null_;  
length y $4;  
/* 220x is the location of the ASCB pointer */  
x=220x;  
y=peekc(peek(x));  
put 'y= ' y;  
run;
```

**Example 2: Creating a DATA Step View**  
This example, specific to the z/OS operating environment, also uses both the PEEK and PEEKC functions. It creates a DATA step view that accesses the entries in the Task Input Output Table (TIOT). The PRINT procedure is then used to print the entries. Entries in the TIOT include the three components outlined in the following list. In this example, TIOT represents the starting address of the TIOT entry.

| TIOT+4   | is the DDname. This component takes up 8 bytes. |
| TIOT+12  | is a 3-byte pointer to the Job File Control Block (JFCB). |
| TIOT+134 | is the volume serial number (volser) of the data set. This component takes up 6 bytes. |

Here is the program:

```sas
/* Create a DATA step view of the contents of the TIOT. The code steps through each TIOT entry to extract the DDname, JFCB, and volser of each DDname that has been allocated for the current task. The data set name is also extracted from the JFCB. */

data save.tiot/view=save.tiot;
  length ddname $8 volser $6 dsname $44;
  /* Get the TCB (Task Control Block) address */
```
functions and call routines

/* from the PSATOLD variable in the PSA */
/* (Prefixed Save Area). The address of */
/* the PSA is 21CX. Add 12 to the address */
/* of the TCB to get the address of the */
/* Tiot. Add 24 to bypass the 24-byte */
/* header, so that TIOTVAR represents the */
/* start of the TIOT entries. */

tiotvar=peek(peek(021CX)+12)+24;
/* Loop through all TIOT entries until the */
/* TIOT entry length (indicated by the */
/* value of the first byte) is 0. */

do while(peek(tiotvar,1));
/* Check to see whether the current TIOT */
/* entry is a freed TIOT entry (indicated */
/* by the high order bit of the second */
/* byte of the TIOT entry). If it is not */
/* freed, then proceed. */

if peek(tiotvar+1,1)NE'1.......‘B then do;
ddbname=peekc(tiotvar+4);
jfcb=peek(tiotvar+12,3);
volser=peekc(jfcb+134);
/* Add 16 to the JFCB value to get */
/* the data set name. The data set */
/* name is 44 bytes. */

dname=peekc(jfcb+16);
output;
end;
/* Increment the TIOTVAR value to point */
/* to the next TIOT entry. This is done */
/* by adding the length of the current */
/* TIOT entry (indicated by first byte */
/* of the entry) to the current value */
/* of TIOTVAR. */

end;
/* The final DATA step view does not */
/* contain the TIOTVAR and JFCB variables. */

keep ddbname volser dname;
run;
/* Print the TIOT entries. */
proc print data=save.tiot uniform width=minimum;
run;
In the PROC PRINT statement, the UNIFORM option ensures that each page of the output is formatted exactly the same way. WIDTH=MINIMUM causes the PRINT procedure to use the minimum column width for each variable on the page. The column width is defined by the longest data value in that column.

See Also

CALL Routine:
- “CALL POKE Routine” on page 352
Functions:
- “ADDR Function” on page 295
- “PEEK Function” on page 708

PEEKCLONG Function

Stores the contents of a memory address in a character variable on 32-bit and 64-bit platforms

Category: Special
See: PEEKCLONG Function in the documentation for your operating environment.

Syntax

PEEKCLONG(address,<length>)

Arguments

address
specifies a character string that is the pointer address.

length
specifies the length of the character data.

Default: 8
Range: 1 to 32,767

Details

If you do not have access to the memory storage location that you are requesting, the PEEKCLONG function returns an “Invalid argument” error.

Comparisons

The PEEKCLONG function stores the contents of a memory address in a character variable.

The PEEKLONG function stores the contents of a memory address in a numeric variable. It assumes that the input address refers to an integer in memory.
Examples

Example 1: Example for a 32-bit Platform  
The following example returns the pointer address for the character variable Z.

```sas
data _null_;  
x='ABCDE';  
y=addrlong(x);  
z=peekclong(y,2);  
put z=;  
run;
```

The output from the SAS log is:  z=\texttt{AB}

Example 2: Example for a 64-bit Platform  
The following example, specific to the z/OS operating environment, returns the pointer address for the character variable Y.

```sas
data _null_;  
length y $4;  
x220addr=put(220x,pib4.);  
ascb=peeklong(x220addr);  
ascbaddr=put(ascb,pib4.);  
y=peekclong(ascbaddr);  
run;
```

The output from the SAS log is:  y='\texttt{ASCB}'

See Also

Function:  
“PEEKLONG Function” on page 713

PEEKLONG Function

Stores the contents of a memory address in a numeric variable on 32-bit and 64-bit platforms

Category:  Special

See:  PEEKLONG Function in the documentation for your operating environment

Syntax

```sas
PEEKLONG(address<,length>)
```

Arguments

`address`
specifies a character string that is the memory address.

`length`
specifies the length of the character data.
**Default:** 4 on 32-bit machines; 8 on 64-bit machines.

**Range:** 1-4 on 32-bit machines; 1-8 on 64-bit machines.

**Details**

If you don’t have access to the memory storage location that you are requesting, the PEEKLONG function returns an “Invalid argument” error.

**Comparisons**

The PEEKLONG function stores the contents of a memory address in a numeric variable. It assumes that the input address refers to an integer in memory.

The PEEKCLONG function stores the contents of a memory address in a character variable. It assumes that the input address refers to character data.

**Examples**

**Example 1: Example for a 32-bit Platform**

The following example returns the pointer address for the numeric variable Z.

```sas
data _null_;  
  length y $4;  
  y=put(1,IB4.);  
  addry=addrlong(y);  
  z=peeklong(addry,4);  
  put z=;  
run;
```

The output from the SAS log is: `z=1`

**Example 2: Example for a 64-bit Platform**

The following example, specific to the z/OS operating environment, returns the pointer address for the numeric variable X.

```sas
data _null_;  
  x=peeklong(put(16,pib4.));  
  put x=hex8.;  
run;
```

The output from the SAS log is: `x=00FCFCB0`

**See Also**

Function:

“PEEKCLONG Function” on page 712

---

**PERM Function**

Computes the number of permutations of $n$ items taken $r$ at a time

**Category:** Mathematical
Syntax

PERM(n<r>)

Arguments

\( n \)

is an integer that represents the total number of elements from which the sample is chosen.

\( r \)

is an optional integer value that represents the number of chosen elements. If \( r \) is omitted, the function returns the factorial of \( n \).

Restriction: \( r \leq n \)

Details

The mathematical representation of the PERM function is given by the following equation:

\[
PERM (n, r) = \frac{n!}{(n-r)!}
\]

with \( n \geq 0 \), \( r \geq 0 \), and \( n \geq r \).

If the expression cannot be computed, a missing value is returned.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=perm(5,1);</td>
<td>5</td>
</tr>
<tr>
<td>x=perm(5);</td>
<td>120</td>
</tr>
<tr>
<td>x=perm(5,2)</td>
<td>20</td>
</tr>
</tbody>
</table>

See Also

Functions:

“COMB Function” on page 442
“FACT Function” on page 516

POINT Function

Locates an observation identified by the NOTE function
Category: SAS File I/O

Syntax

POINT(data-set-id,note-id)

Arguments

data-set-id
specifies the data set identifier that the OPEN function returns.

note-id
specifies the identifier assigned to the observation by the NOTE function.

Details

POINT returns 0 if the operation was successful, ≠0 if it was not successful. POINT prepares the program to read from the SAS data set. The Data Set Data Vector is not updated until a read is done using FETCH or FETCHOBS.

Examples

This example calls NOTE to obtain an observation ID for the most recently read observation of the SAS data set MYDATA. It calls POINT to point to that observation, and calls FETCH to return the observation marked by the pointer.

```sas
%let dsid=%sysfunc(open(mydata,i));
%let rc=%sysfunc(fetch(&dsid));
%let noteid=%sysfunc(note(&dsid));
...more macro statements...
%let rc=%sysfunc(point(&dsid,&noteid));
%let rc=%sysfunc(fetch(&dsid));
...more macro statements...
%let rc=%sysfunc(close(&dsid));
```

See Also

Functions:

“DROPNOTE Function” on page 507
“NOTE Function” on page 671
“OPEN Function” on page 690

POISSON Function

Returns the probability from a Poisson distribution

Category: Probability

See: “CDF Function” on page 416
Syntax
POISSON(m,n)

Arguments

m  
is a numeric mean parameter.  
Range:  m ≥ 0

n  
is an integer random variable.  
Range:  n ≥ 0

Details
The POISSON function returns the probability that an observation from a Poisson distribution, with mean m, is less than or equal to n. To compute the probability that an observation is equal to a given value, n, compute the difference of two probabilities from the Poisson distribution for n and n− .

Examples

SAS Statements  |  Results
---|---
x=poisson(1,2);  |  0.9196986029

PROBBETA Function

Returns the probability from a beta distribution

Category:  Probability
See:  “CDF Function” on page 416

Syntax
PROBBETA(x,a,b)

Arguments

x  
is a numeric random variable.
Range: $0 \leq x \leq 1$

$a$

is a numeric shape parameter.

Range: $a > 0$

$b$

is a numeric shape parameter.

Range: $b > 0$

Details

The PROBBETA function returns the probability that an observation from a beta distribution, with shape parameters $a$ and $b$, is less than or equal to $x$.

Example

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=probbeta(.2,3,4);</td>
<td>0.09888</td>
</tr>
</tbody>
</table>

PROBBNML Function

Returns the probability from a binomial distribution

Category: Probability

See: “CDF Function” on page 416, “PDF Function” on page 695

Syntax

PROBBNML($p,n,m$)

Arguments

$p$

is a numeric probability of success parameter.

**RANGE:** $0 \leq p \leq 1$

$n$

is an integer number of independent Bernoulli trials parameter.

**RANGE:** $n > 0$

$m$

is an integer number of successes random variable.

**RANGE:** $0 \leq m \leq n$
Details

The PROBBNML function returns the probability that an observation from a binomial distribution, with probability of success \( p \), number of trials \( n \), and number of successes \( m \), is less than or equal to \( m \). To compute the probability that an observation is equal to a given value \( m \), compute the difference of two probabilities from the binomial distribution for \( m \) and \( m-1 \) successes.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=probbnml(0.5,10,4);</td>
<td>0.376953125</td>
</tr>
</tbody>
</table>

PROBGNRM Function

Computes a probability from the bivariate normal distribution

Category: Probability

Syntax

\[
PROBGNRM(x, y, r)
\]

Arguments

\( x \)

is a numeric variable.

\( y \)

is a numeric variable.

\( r \)

is a numeric correlation coefficient.

Range: \(-1 \leq r \leq 1\)
**Details**

The PROBBNRM function returns the probability that an observation \((X, Y)\) from a standardized bivariate normal distribution with mean 0, variance 1, and a correlation coefficient \(r\), is less than or equal to \((x, y)\). That is, it returns the probability that \(X \leq x\) and \(Y \leq y\). The following equation describes the PROBBNRM function, where \(u\) and \(v\) represent the random variables \(x\) and \(y\), respectively:

\[
\text{PROBBNRM}(x, y, r) = \frac{1}{2\pi\sqrt{1 - r^2}} \int_{-\infty}^{x} \int_{-\infty}^{y} \exp\left(-\frac{u^2 - 2ruv + v^2}{2(1 - r^2)}\right) dv \, du
\]

**Examples**

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>p=probbnrm(.4, -.3, .2); put p;</td>
<td>0.2783183345</td>
</tr>
</tbody>
</table>

**PROBCHI Function**

Returns the probability from a chi-squared distribution

**Category:** Probability

**See:** “CDF Function” on page 416

**Syntax**

\[
\text{PROBCHI}(x, df<, nc>)
\]

**Arguments**

- \(x\)
  - is a numeric random variable.
  - **Range:** \(x \geq 0\)

- \(df\)
  - is a numeric degrees of freedom parameter.
  - **Range:** \(df > 0\)

- \(nc\)
  - is an optional numeric noncentrality parameter.
  - **Range:** \(nc \geq 0\)
Details
The PROBCHI function returns the probability that an observation from a chi-square
distribution, with degrees of freedom \( df \) and noncentrality parameter \( nc \), is less than or
equal to \( x \). This function accepts a noninteger degrees of freedom parameter \( df \). If the
optional parameter \( nc \) is not specified or has the value 0, the value returned is from the
central chi-square distribution.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = \text{probchi}(11.264,11) );</td>
<td>0.5785813293</td>
</tr>
</tbody>
</table>

PROBF Function

Returns the probability from an \( F \) distribution

Category: Probability

See: “CDF Function” on page 416

Syntax

\[
\text{PROBF}(x, ndf, ddf, <nc>)
\]

Arguments

- \( x \) is a numeric random variable.
  - Range: \( x \geq 0 \)
- \( ndf \) is a numeric numerator degrees of freedom parameter.
  - Range: \( ndf > 0 \)
- \( ddf \) is a numeric denominator degrees of freedom parameter.
  - Range: \( ddf > 0 \)
- \( nc \) is an optional numeric noncentrality parameter.
  - Range: \( nc \geq 0 \)

Details
The PROBF function returns the probability that an observation from an \( F \) distribution,
with numerator degrees of freedom \( ndf \), denominator degrees of freedom \( ddf \), and
noncentrality parameter $nc$, is less than or equal to $x$. The PROBF function accepts noninteger degrees of freedom parameters $ndf$ and $ddf$. If the optional parameter $nc$ is not specified or has the value 0, the value returned is from the central $F$ distribution.

The significance level for an $F$ test statistic is given by

$$p = 1 - \text{probf}(x, ndf, ddf);$$

**Examples**

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = \text{probf}(3.32, 2, 3);$</td>
<td>0.8263933602</td>
</tr>
</tbody>
</table>

---

**PROBGAM Function**

Returns the probability from a gamma distribution

**Category:** Probability

**See:** “CDF Function” on page 416

**Syntax**

PROBGAM($x,a$)

**Arguments**

$x$

is a numeric random variable.

Range: $x \geq 0$

$a$

is a numeric shape parameter.

Range: $a > 0$

**Details**

The PROBGAM function returns the probability that an observation from a gamma distribution, with shape parameter $a$, is less than or equal to $x$.

**Examples**

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = \text{probgam}(1, 3);$</td>
<td>0.0803013971</td>
</tr>
</tbody>
</table>
PROBHYPR Function

Returns the probability from a hypergeometric distribution

Category: Probability
See: “CDF Function” on page 416

Syntax

PROBHYPR(N,K,n,x<r>)

Arguments

N
is an integer population size parameter, with \( N \geq 1 \).
Range: 0 ≤ N

K
is an integer number of items in the category of interest parameter.
Range: 0 ≤ K ≤ N

n
is an integer sample size parameter.
Range: 0 ≤ n ≤ N

x
is an integer random variable.
Range: max(0, K + n−N) ≤ x ≤ min(K,n)

r
is an optional numeric odds ratio parameter.
Range: r ≥ 0

Details

The PROBHYPR function returns the probability that an observation from an extended hypergeometric distribution, with population size \( N \), number of items \( K \), sample size \( n \), and odds ratio \( r \), is less than or equal to \( x \). If the optional parameter \( r \) is not specified or is set to 1, the value returned is from the usual hypergeometric distribution.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=probhypr(200,50,10,2);</td>
<td>0.5236734081</td>
</tr>
</tbody>
</table>
**PROBIT Function**

Returns a quantile from the standard normal distribution

Category: Quantile

**Syntax**

PROBIT\( (p) \)

**Arguments**

\( p \)

is a numeric probability.

**Range:** \( 0 < p < 1 \)

**Details**

The PROBIT function returns the \( p \)th quantile from the standard normal distribution. The probability that an observation from the standard normal distribution is less than or equal to the returned quantile is \( p \).

**CAUTION:**

The result could be truncated to lie between -8.222 and 7.941.

**Note:** PROBIT is the inverse of the PROBNORM function.

**Examples**

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=probit(.025);</td>
<td>-1.959963985</td>
</tr>
<tr>
<td>x=probit(1.e-7);</td>
<td>-5.199337582</td>
</tr>
</tbody>
</table>

**PROBMC Function**

Computes a probability or a quantile from various distributions for multiple comparisons of means

Category: Probability

**Syntax**

PROBMC\( (\text{distribution}, q, \text{prob}, df, nparms<, parameters>) \)
Arguments

distribution
is a character string that identifies the distribution. Valid distributions are

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>One-sided Dunnett</td>
<td>‘DUNNETT1’</td>
</tr>
<tr>
<td>Two-sided Dunnett</td>
<td>‘DUNNETT2’</td>
</tr>
<tr>
<td>Maximum Modulus</td>
<td>‘MAXMOD’</td>
</tr>
<tr>
<td>Studentized Range</td>
<td>‘RANGE’</td>
</tr>
<tr>
<td>Williams</td>
<td>‘WILLIAMS’</td>
</tr>
</tbody>
</table>

q
is the quantile from the distribution.
Restriction: Either q or prob can be specified, but not both.

prob
is the left probability from the distribution.
Restriction: Either prob or q can be specified, but not both.

df
is the degrees of freedom.
Note: A missing value is interpreted as an infinite value. △
nparms
is the number of treatments.
Note: For DUNNETT1 and DUNNETT2, the control group is not counted. △

parameters
is an optional set of nparms parameters that must be specified to handle the case of unequal sample sizes. The meaning of parameters depends on the value of distribution. If parameters is not specified, equal sample sizes are assumed; this is usually the case for a null hypothesis.

Details
The PROBMC function returns the probability or the quantile from various distributions with finite and infinite degrees of freedom for the variance estimate.
The prob argument is the probability that the random variable is less than q. Therefore, p-values can be computed as 1− prob. For example, to compute the critical value for a 5% significance level, set prob = 0.95. The precision of the computed probability is O(10^-5) (absolute error); the precision of computed quantile is O(10^-5).

Note: The studentized range is not computed for finite degrees of freedom and unequal sample sizes. △

Note: Williams’ test is computed only for equal sample sizes. △

Formulas and Parameters The equations listed here define expressions used in equations that relate the probability, prob, and the quantile, q, for different
distributions and different situations within each distribution. For these equations, let \( \nu \) be the degrees of freedom, \( df \).

\[
d\mu_\nu (x) = \frac{\nu^{\frac{\nu}{2}}}{\Gamma(\frac{\nu}{2}) 2^{\frac{\nu}{2}-1}} x^{\nu-1} e^{-\frac{x^2}{2}} dx
\]

\[
\phi (x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}
\]

\[
\Phi (x) = \int_{-\infty}^{x} \phi (u) du
\]

**Many-One \( t \)-Statistics: Dunnett’s One-Sided Test**

- This case relates the probability, \( prob \), and the quantile, \( q \), for the unequal case with finite degrees of freedom. The \textit{parameters} are \( \lambda_1, ..., \lambda_k \), the value of \textit{nparms} is set to \( k \), and the value of \( df \) is set to \( \nu \). The equation follows:

\[
prob = \int_{0}^{\infty} \int_{-\infty}^{\infty} \phi (y) \prod_{i=1}^{k} \Phi \left( \frac{\lambda_i y + qx}{\sqrt{1 - \lambda_i^2}} \right) dy \ d\mu_\nu (x)
\]

- This case relates the probability, \( prob \), and the quantile, \( q \), for the equal case with finite degrees of freedom. No \textit{parameters} are passed \( \left( \lambda = \sqrt{\frac{1}{2}} \right) \), the value of \textit{nparms} is set to \( k \), and the value of \( df \) is set to \( \nu \). The equation follows:

\[
prob = \int_{0}^{\infty} \int_{-\infty}^{\infty} \phi (y) \left[ \Phi \left( y + \sqrt{2qx} \right) \right]^{k} dy \ d\mu_\nu (x)
\]

- This case relates the probability, \( prob \), and the quantile, \( q \), for the unequal case with infinite degrees of freedom. The \textit{parameters} are \( \lambda_1, ..., \lambda_k \), the value of \textit{nparms} is set to \( k \), and the value of \( df \) is set to missing. The equation follows:

\[
prob = \int_{-\infty}^{\infty} \phi (y) \prod_{i=1}^{k} \Phi \left( \frac{\lambda_i y + q}{\sqrt{1 - \lambda_i^2}} \right) dy
\]

- This case relates the probability, \( prob \), and the quantile, \( q \), for the equal case with infinite degrees of freedom. No \textit{parameters} are passed \( \left( \lambda = \sqrt{\frac{1}{2}} \right) \), the value of \textit{nparms} is set to \( k \), and the value of \( df \) is set to missing. The equation follows:
Many-One t-Statistics: Dunnett’s Two-sided Test

This case relates the probability, \( \text{prob} \), and the quantile, \( q \), for the unequal case with finite degrees of freedom. The parameters are \( \lambda_1, \ldots, \lambda_k \), the value of \( \text{nparms} \) is set to \( k \), and the value of \( df \) is set to \( \nu \). The equation follows:

\[
\text{prob} = \int_{-\infty}^{\infty} \phi(y) \left[ \Phi\left( y + \sqrt{2q} \right) \right]^k dy
\]

This case relates the probability, \( \text{prob} \), and the quantile, \( q \), for the equal case with finite degrees of freedom. No parameters are passed, the value of \( \text{nparms} \) is set to \( k \), and the value of \( df \) is set to \( \nu \). The equation follows:

\[
\text{prob} = \int_{0}^{\infty} \int_{-\infty}^{\infty} \phi(y) \prod_{i=1}^{k} \left[ \Phi\left( \frac{\lambda_i y + q x}{\sqrt{1 - \lambda_i^2}} \right) - \Phi\left( \frac{\lambda_i y - q x}{\sqrt{1 - \lambda_i^2}} \right) \right] dy du_{\nu}(x)
\]

This case relates the probability, \( \text{prob} \), and the quantile, \( q \), for the unequal case with infinite degrees of freedom. The parameters are \( \lambda_1, \ldots, \lambda_k \), the value of \( \text{nparms} \) is set to \( k \), and the value of \( df \) is set to missing. The equation follows:

\[
\text{prob} = \int_{-\infty}^{\infty} \phi(y) \prod_{i=1}^{k} \left[ \Phi\left( \frac{\lambda_i y + q}{\sqrt{1 - \lambda_i^2}} \right) - \Phi\left( \frac{\lambda_i y - q}{\sqrt{1 - \lambda_i^2}} \right) \right] dy
\]

This case relates the probability, \( \text{prob} \), and the quantile, \( q \), for the equal case with infinite degrees of freedom. No parameters are passed, the value of \( \text{nparms} \) is set to \( k \), and the value of \( df \) is set to missing. The equation follows:

\[
\text{prob} = \int_{-\infty}^{\infty} \phi(y) \left[ \Phi\left( y + \sqrt{2q} \right) - \Phi\left( y - \sqrt{2q} \right) \right]^k dy
\]

The Studentized Range

Note: The studentized range is not computed for finite degrees of freedom and unequal sample sizes. △
This case relates the probability, \( \text{prob} \), and the quantile, \( q \), for the equal case with finite degrees of freedom. No parameters are passed, the value of \( \text{nparms} \) is set to \( k \), and the value of \( df \) is set to \( \nu \). The equation follows:

\[
\text{prob} = \int_0^\infty \int_{-\infty}^\infty k \phi(y) \left[ \Phi\left( y \right) - \Phi\left( y - qx \right) \right]^{k-1} dy \, d\mu_\nu(x)
\]

This case relates the probability, \( \text{prob} \), and the quantile, \( q \), for the unequal case with infinite degrees of freedom. The parameters are \( \sigma_1, \ldots, \sigma_k \), the value of \( \text{nparms} \) is set to \( k \), and the value of \( df \) is set to missing. The equation follows:

\[
\text{prob} = \int_{-\infty}^\infty \sum_{j=1}^k \left\{ \prod_{i=1}^k \left[ \Phi\left( \frac{y}{\sigma_i} \right) - \Phi\left( \frac{y - qx}{\sigma_i} \right) \right] \right\} \phi\left( \frac{y}{\sigma_j} \right) \frac{1}{\sigma_j} dy
\]

This case relates the probability, \( \text{prob} \), and the quantile, \( q \), for the equal case with infinite degrees of freedom. No parameters are passed, the value of \( \text{nparms} \) is set to \( k \), and the value of \( df \) is set to missing. The equation follows:

\[
\text{prob} = \int_{-\infty}^\infty k \phi(y) \left[ \Phi\left( y \right) - \Phi\left( y - qx \right) \right]^{k-1} dy
\]

The Studentized Maximum Modulus

This case relates the probability, \( \text{prob} \), and the quantile, \( q \), for the unequal case with finite degrees of freedom. The parameters are \( \sigma_1, \ldots, \sigma_k \), the value of \( \text{nparms} \) is set to \( k \), and the value of \( df \) is set to \( \nu \). The equation follows:

\[
\text{prob} = \int_0^\infty \prod_{i=1}^k \left[ 2 \Phi\left( \frac{qx}{\sigma_i} \right) - 1 \right] d\mu_\nu(x)
\]

This case relates the probability, \( \text{prob} \), and the quantile, \( q \), for the equal case with finite degrees of freedom. No parameters are passed, the value of \( \text{nparms} \) is set to \( k \), and the value of \( df \) is set to \( \nu \). The equation follows:

\[
\text{prob} = \int_0^\infty \left[ 2 \Phi\left( qx \right) - 1 \right]^k d\mu_\nu(x)
\]

This case relates the probability, \( \text{prob} \), and the quantile, \( q \), for the unequal case with infinite degrees of freedom. The parameters are \( \sigma_1, \ldots, \sigma_k \), the value of \( \text{nparms} \) is set to \( k \), and the value of \( df \) is set to missing. The equation follows:
This case relates the probability, \( \text{prob} \), and the quantile, \( q \), for the equal case with infinite degrees of freedom. No \textit{parameters} are passed, the value of \( \text{nparms} \) is set to \( k \), and the value of \( \text{df} \) is set to missing. The equation follows:

\[
\text{prob} = [2\Phi(q) - 1]^k
\]

\textbf{Williams’ Test} \quad \text{PROBMC computes the probabilities or quantiles from the distribution defined in Williams (1971, 1972) (See “References” on page 928). It arises when you compare the dose treatment means with a control mean to determine the lowest effective dose of treatment.}

\textit{Note:} Williams’ Test is computed only for equal sample sizes. \( \Delta \)

Let \( X_1, X_2, \ldots, X_k \) be identical independent N(0,1) random variables. Let \( \bar{Y}_k \) denote their average given by

\[
\bar{Y}_k = \frac{X_1 + X_2 + \ldots + X_k}{k}
\]

It is required to compute the distribution of

\[
(Y_k - Z) / S
\]

where

- \( Y_k \) is as defined previously
- \( Z \) is a N(0,1) independent random variable
- \( S \) is such that \( \frac{1}{2} \nu S^2 \) is a \( \chi^2 \) variable with \( \nu \) degrees of freedom.

As described in Williams (1971) (See “References” on page 928), the full computation is extremely lengthy and is carried out in three stages.

1. Compute the distribution of \( Y_k \). It is the fundamental (expensive) part of this operation and it can be used to find both the density and the probability of \( Y_k \). Let \( U_i \) be defined as

\[
U_i = \frac{X_1 + X_2 + \ldots + X_i}{i}, \quad i = 1, 2, \ldots, k
\]

You can write a recursive expression for the probability of \( Y_i > d \), with \( d \) being any real number.
To compute this probability, start from a \( N(0,1) \) density function and recursively compute the convolution

\[
D(U_1 = x) = \phi(x)
\]

and recursively compute the convolution

\[
D(U_k = x, U_{k-1} < d, \ldots, U_1 < d) = \\
\int_{-\infty}^{d} D(U_{k-1} = y, U_{k-2} < d, \ldots, U_1 < d) (k-1) \phi(kx - (k-1)y) dy
\]

From this sequential convolution, it is possible to compute all the elements of the recursive equation for \( \Pr(Y_k < d) \), shown previously.

2 Compute the distribution of \( Y_k - Z \). This involves another convolution to compute the probability

\[
\Pr((Y_k - Z) > d) = \int_{-\infty}^{\infty} \Pr(Y_k > \sqrt{2d} + y) \phi(y) dy
\]

3 Compute the distribution of \( (Y_k - Z)/S \). This involves another convolution to compute the probability

\[
\Pr((Y_k - Z)/S > t) = \int_{0}^{\infty} \Pr((Y_k - Z)/S > ty) d\mu(y)
\]

The third stage is not needed when \( \nu = \infty \). Due to the complexity of the operations, this lengthy algorithm is replaced by a much faster one when \( k \leq 15 \) for both finite and infinite degrees of freedom \( \nu \). For \( k \geq 16 \), the lengthy computation is carried out. It is extremely expensive and very slow due to the complexity of the algorithm.

**Comparisons**

The MEANS statement in the GLM Procedure of SAS/STAT Software computes the following tests:

- Dunnett’s one-sided test
Dunnett’s two-sided test
Studentized Range.

**Examples**

**Example 1: Using PROBMC to Compute Probabilities**  
This example shows how to use PROBMC in a DO loop to compute probabilities:

```sas
data probs;
array par{5};
   par{1}=.5;
   par{2}=.51;
   par{3}=.55;
   par{4}=.45;
   par{5}=.2;
   df=40;
   q=1;
   do test="dunnett1","dunnett2", "maxmod";
      prob=probmc(test, q, ., df, 5, of par1--par5);
      put test $10. df q e18.13 prob e18.13;
   end;
run;
```

Output 4.30 shows the results of this DATA step that are printed to the SAS log.

**Output 4.30  Probabilities from PROBMC**

<table>
<thead>
<tr>
<th>Test</th>
<th>df</th>
<th>q</th>
<th>Prob</th>
</tr>
</thead>
<tbody>
<tr>
<td>DUNNETT1</td>
<td>40</td>
<td>1.00000000000000E+00 4.82992188740E-01</td>
<td></td>
</tr>
<tr>
<td>DUNNETT2</td>
<td>40</td>
<td>1.00000000000000E+00 1.64023099613E-01</td>
<td></td>
</tr>
<tr>
<td>MAXMOD</td>
<td>40</td>
<td>1.00000000000000E+00 8.02784203408E-01</td>
<td></td>
</tr>
</tbody>
</table>

**Example 2: Comparing Means**  
This example shows how to compare group means to find where the significant differences lie. The data for this example is taken from a paper by Duncan (1955) (See “References” on page 928) and can also be found in Hochberg and Tamhane (1987) (See “References” on page 928). The group means are

49.6  
71.2  
67.6  
61.5  
71.3  
58.1  
61.0

For this data, the mean square error is $s^2 = 79.64$ ($s = 8.924$) with $\nu = 30$.

```sas
data duncan;
   array tr{7}$;
   array mu{7};
   n=7;
   do i=1 to n;
      input tr{i} $1. mu{i};
   end;
   input df s alpha;
   prob= 1--alpha;
```
/* compute the interval */
x = probmc("RANGE", ., prob, df, 7);
w = x * s / sqrt(6);
/* compare the means */
do i = 1 to n;
do j = i + 1 to n;
dmean = abs(mu{i} - mu{j});
if dmean >= w then do;
    put tr{i} tr{j} dmean;
end;
end;
datalines;
A 49.6
B 71.2
C 67.6
D 61.5
E 71.3
F 58.1
G 61.0
30 8.924 .05
;

Output 4.31 shows the results of this DATA step that are printed to the SAS log.

Output 4.31  Group Differences

A B 21.6
A C 18
A E 21.7

Example 3: Computing Confidence Intervals  This example shows how to compute 95% one-sided and two-sided confidence intervals of Dunnett’s test. This example and the data come from Dunnett (1955) (See “References” on page 928) and can also be found in Hochberg and Tamhane (1987) (See “References” on page 928). The data are blood count measurements on three groups of animals. As shown in the following table, the third group serves as the control, while the first two groups were treated with different drugs. The numbers of animals in these three groups are unequal.

<table>
<thead>
<tr>
<th>Treatment Group</th>
<th>Drug A</th>
<th>Drug B</th>
<th>Control</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>9.76</td>
<td>12.80</td>
<td>7.40</td>
</tr>
<tr>
<td>8.80</td>
<td>9.68</td>
<td>8.50</td>
<td></td>
</tr>
<tr>
<td>7.68</td>
<td>12.16</td>
<td>7.20</td>
<td></td>
</tr>
<tr>
<td>9.36</td>
<td>9.20</td>
<td>8.24</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10.55</td>
<td>9.84</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>8.32</td>
<td></td>
</tr>
<tr>
<td>Group Mean</td>
<td>8.90</td>
<td>10.88</td>
<td>8.25</td>
</tr>
<tr>
<td>n</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
The mean square error \( s^2 = 1.3805 \) (\( s = 1.175 \)) with \( \nu = 12 \).

data a;
  array drug{3}$;
  array count{3};
  array mu{3};
  array lambda{2};
  array delta{2};
  array left{2};
  array right{2};

  /* input the table */
do i = 1 to 3;
  input drug{i} count{i} mu{i};
end;

  /* input the alpha level, */
  /* the degrees of freedom, */
  /* and the mean square error */
  input alpha df s;

  /* from the sample size, */
  /* compute the lambdas */
do i = 1 to 2;
  lambda{i} = sqrt(count{i}/(count{i} + count{3}));
end;

  /* run the one-sided Dunnett's test */
test="dunnett1";
  x = probmc(test,.,1-alpha,df,2,of lambda1--lambda2);
do i = 1 to 2;
  delta{i} = x * s * sqrt(1/count{i} + 1/count{3});
  left{i} = mu{i} - mu{3} - delta{i};
end;
  put test $10. x left{1} left{2};

  /* run the two-sided Dunnett's test */
test="dunnett2";
  x = probmc(test,.,1-alpha,df,2,of lambda1--lambda2);
do i=1 to 2;
  delta{i} = x * s * sqrt(1/count{i} + 1/count{3});
  left{i} = mu{i} - mu{3} - delta{i};
  right{i} = mu{i} - mu{3} + delta{i};
end;
  put test $10. left{1} right{1};
  put test $10. left{2} right{2};
datalines;
A 4 8.90
B 5 10.88
C 6 8.25
Output 4.32 shows the results of this DATA step that are printed to the SAS log.

Output 4.32  Confidence Intervals

<table>
<thead>
<tr>
<th>Dunnett1</th>
<th>2.1210786586</th>
<th>-0.958751705</th>
<th>1.1208571303</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dunnett2</td>
<td>-1.256411895</td>
<td>2.5564118953</td>
<td></td>
</tr>
<tr>
<td>Dunnett2</td>
<td>0.8416271203</td>
<td>4.4183728797</td>
<td></td>
</tr>
</tbody>
</table>

**Example 4: Computing Williams' Test**  
Suppose that a substance has been tested at seven levels in a randomized block design of eight blocks. The observed treatment means are as follows:

<table>
<thead>
<tr>
<th>Treatment</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>X₀</td>
<td>10.4</td>
</tr>
<tr>
<td>X₁</td>
<td>9.9</td>
</tr>
<tr>
<td>X₂</td>
<td>10.0</td>
</tr>
<tr>
<td>X₃</td>
<td>10.6</td>
</tr>
<tr>
<td>X₄</td>
<td>11.4</td>
</tr>
<tr>
<td>X₅</td>
<td>11.9</td>
</tr>
<tr>
<td>X₆</td>
<td>11.7</td>
</tr>
</tbody>
</table>

The mean square, with \((7 - 1)(8 - 1) = 42\) degrees of freedom, is \(s^2 = 1.16\). 
Determine the maximum likelihood estimates \(M_i\) through the averaging process.

- Because \(X₀ > X₁\), form \(X₀₁ = (X₀ + X₁)/2 = 10.15\).
- Because \(X₀₁ > X₂\), form \(X₀₁₂ = (X₀ + X₁ + X₂)/3 = (2X₀₁ + X₂)/3 = 10.1\).
- \(X₀₁₂ < X₃ < X₄ < X₅\)
- Because \(X₅ > X₆\), form \(X₅₆ = (X₅ + X₆)/2 = 11.8\).

Now the order restriction is satisfied.

The maximum likelihood estimates under the alternative hypothesis are

\[
\begin{align*}
M₀ &= M₁ = M₂ = X₀₁₂ = 10.1 \\
M₃ &= X₃ = 10.6 \\
M₄ &= X₄ = 11.4 \\
M₅ &= M₆ = X₅₆ = 11.8
\end{align*}
\]

Now compute \(t = (11.8 - 10.4) / \sqrt{2s^2/8} = 2.60\), and the probability that corresponds to \(k = 6\), \(n = 42\), and \(t = 2.60\) is .9924467341, which shows strong evidence that there is a response to the substance. You can also compute the quantiles for the upper 5% and 1% tails, as shown in the following table.
PROBNEGB Function

Returns the probability from a negative binomial distribution

Category: Probability
See: “CDF Function” on page 416

Syntax

PROBNEGB(p,n,m)

Arguments

p
is a numeric probability of success parameter.
Range: \(0 \leq p \leq 1\)

n
is an integer number of successes parameter.
Range: \(n \geq 1\)

m
is a positive integer random variable, the number of failures.
Range: \(m \geq 0\)

Details

The PROBNEGB function returns the probability that an observation from a negative binomial distribution, with probability of success \(p\) and number of successes \(n\), is less than or equal to \(m\).

To compute the probability that an observation is equal to a given value \(m\), compute the difference of two probabilities from the negative binomial distribution for \(m\) and \(m-1\).

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>prob=probnegb(“williams”,2.6,.,42,6);</td>
<td>0.99244673</td>
</tr>
<tr>
<td>quant5=probnegb(“williams”,.,.95,42,6);</td>
<td>1.80654052</td>
</tr>
<tr>
<td>quant1=probnegb(“williams”,.,.99,42,6);</td>
<td>2.49087829</td>
</tr>
<tr>
<td>x=probnegb(0.5,2,1);</td>
<td>0.5</td>
</tr>
</tbody>
</table>
PROBNORM Function

Returns the probability from the standard normal distribution

Category: Probability
See: “CDF Function” on page 416

Syntax
PROBNORM(x)

Arguments

x
is a numeric random variable.

Details
The PROBNORM function returns the probability that an observation from the standard normal distribution is less than or equal to x.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=probnorm(1.96);</td>
<td>0.9750021049</td>
</tr>
</tbody>
</table>

PROBT Function

Returns the probability from a t distribution

Category: Probability
See: “CDF Function” on page 416, “PDF Function” on page 695

Syntax
PROBT(x,df<,nc>)

Arguments

x
is a numeric random variable.
\( df \)

is a numeric degrees of freedom parameter.

**Range:** \( df > 0 \)

\( nc \)

is an optional numeric noncentrality parameter.

### Details

The PROBT function returns the probability that an observation from a Student’s \( t \) distribution, with degrees of freedom \( df \) and noncentrality parameter \( nc \), is less than or equal to \( x \). This function accepts a noninteger degree of freedom parameter \( df \). If the optional parameter, \( nc \), is not specified or has the value 0, the value that is returned is from the central Student’s \( t \) distribution.

The significance level of a two-tailed \( t \) test is given by

\[
p = (1 - \text{probt}(\text{abs}(x), df)) \times 2;
\]

### Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = \text{probt}(0.9, 5) );</td>
<td>0.7953143998</td>
</tr>
</tbody>
</table>

### PROPCASE Function

**Converts all words in an argument to proper case**

**Category:** Character

### Syntax

\[
\text{PROPCASE}(\text{argument } <, \text{delimiter(s)}>)
\]

### Arguments

- **argument**
  - specifies any SAS character expression.

- **delimiter**
  - specifies one or more delimiters that are enclosed in quotation marks. The default delimiters are blank, forward slash, hyphen, open parenthesis, period, and tab.
  - **Tip:** If you use this argument, then the default delimiters, including the blank, are no longer in effect.

### Details

The PROPCASE function copies a character argument and converts all uppercase letters to lowercase letters. It then converts to uppercase the first character of a word.
that is preceded by a blank, forward slash, hyphen, open parenthesis, period, or tab. PROPCASE returns the value that is altered.

If you use the second argument, then the default delimiters are no longer in effect.

Examples

Example 1: Changing the Case of Words

The following example shows how PROPCASE handles the case of words:

```sas
data _null_;  
  input place $ 1-40;  
  name=propcase(place);  
  put name;  
  datalines;  
  INTRODUCTION TO THE SCIENCE OF ASTRONOMY  
  VIRGIN ISLANDS (U.S.)  
  SAINT KITTS/NEVIS  
  WINSTON-SALEM, N.C.  
;  
run;  
```

SAS writes the following output to the log:

```
Introduction To The Science Of Astronomy  
Virgin Islands (U.S.)  
Saint Kitts/Nevis  
Winston-Salem, N.C.  
```

Example 2: Using PROPCASE with Other Functions

This example shows that you can use the PROPCASE function inside other functions:

```sas
data _null_;  
  x=lowcase('THIS IS A DOG');  
  y=propcase(x);  
  z=propcase(lowcase('THIS IS A DOG'));  
  put x=;  
  put y=;  
  put z=;  
run;  
```

SAS writes the following output to the log:

```
x=this is a dog  
y=This Is A Dog  
z=This Is A Dog  
```

Example 3: Using PROPCASE with Other Functions: Another Example

```sas
data _null_;  
  string1='VERY RARE BOOKS';  
  /* PROPCASE converts the words in string1 to proper case. */  
  case1=propcase(string1);  
  put case1=;  
  /* TRANWRD searches string1 for the word VERY and */  
  /* replaces VERY with a blank. PROPCASE converts */  
  /* the words in string1 to proper case. */  
```
case2=propcase(tranwrd(string1, 'VERY', ' '));
put case2=
run;

SAS writes the following output to the log:

    case1=Very Rare Books
    case2=Rare Books

**Example 4: Using PROPCASE with a Second Argument**  The following example uses a blank, a hyphen and a single quotation mark as the second argument so that names such as O'Keeffe and Burne-Jones are written correctly.

```sas
options pageno=1 nodate ls=80 ps=64;
data names;
  infile datalines dlm='#';
  input CommonName : $20. CapsName : $20.;
  PropcaseName=propcase(capsname, " -‘");
datalines;
Delacroix, Eugene# EUGENE DELACROIX
O’Keeffe, Georgia# GEORGIA O’KEEFFE
Rockwell, Norman# NORMAN ROCKWELL
Burne-Jones, Edward# EDWARD BURNE-JONES
;
proc print data=names noobs;
  title 'Names of Artists';
run;
```

**Output 4.33**  Output Showing the Results of Using PROPCASE with a Second Argument

<table>
<thead>
<tr>
<th>CommonName</th>
<th>CapsName</th>
<th>PropcaseName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delacroix, Eugene</td>
<td>EUGENE DELACROIX</td>
<td>Eugene Delacroix</td>
</tr>
<tr>
<td>O’Keeffe, Georgia</td>
<td>GEORGIA O’KEEFFE</td>
<td>Georgia O’Keeffe</td>
</tr>
<tr>
<td>Rockwell, Norman</td>
<td>NORMAN ROCKWELL</td>
<td>Norman Rockwell</td>
</tr>
<tr>
<td>Burne-Jones, Edward</td>
<td>EDWARD BURNE-JONES</td>
<td>Edward Burne-Jones</td>
</tr>
</tbody>
</table>

**See Also**

Functions:

- “UPCASE Function” on page 867
- “LOWCASE Function” on page 640

**PRXCHANGE Function**

Performs a pattern-matching replacement
Category: Character String Matching

Syntax

PRXCHANGE(perl-regular-expression | regular-expression-id, times, source)

Arguments

perl-regular-expression
specifies a Perl regular expression.

regular-expression-id
specifies a numeric pattern identifier that is returned from the PRXPARSE function.

Restriction: If you use this argument, you must also use the PRXPARSE function.

times
is a numeric value that specifies the number of times to search for a match and replace a matching pattern.

Tip: If the value of times is –1, then matching patterns continue to be replaced until the end of source is reached.

source
specifies the character expression that you want to search.

Details

The Basics
If you use regular-expression-id, the PRXCHANGE function searches the variable source with the regular-expression-id that is returned by PRXPARSE. It returns the value in source with the changes that were specified by the regular expression. If there is no match, PRXCHANGE returns the unchanged value in source.

If you use perl-regular-expression, PRXMATCH searches the variable source with the perl-regular-expression, and you do not need to call PRXPARSE. You can use PRXMATCH with a perl-regular-expression in a WHERE clause and in PROC SQL.

For more information about pattern matching, see “Pattern Matching Using SAS Regular Expressions (RX) and Perl Regular Expressions (PRX)” on page 260.

Compiling a Perl Regular Expression
If perl-regular-expression is a constant or if it uses the /o option, then the Perl regular expression is compiled once and each use of PRXCHANGE reuses the compiled expression. If perl-regular-expression is not a constant and if it does not use the /o option, then the Perl regular expression is recompiled for each call to PRXCHANGE.

Note: The compile-once behavior occurs when you use PRXCHANGE in a DATA step, in a WHERE clause, or in PROC SQL. For all other uses, the perl-regular-expression is recompiled for each call to PRXCHANGE.

Comparisons
The PRXCHANGE function is similar to the CALL PRXCHANGE routine except that the function returns the value of the pattern-matching replacement as a return argument instead of as one of its parameters.

The Perl regular expression (PRX) functions and CALL routines work together to manipulate strings that match patterns. To see a list and short description of these
functions and CALL routines, see the Character String Matching category in “Functions and CALL Routines by Category” on page 270.

Examples

Example 1: Changing the Order of First and Last Names

Changing the Order of First and Last Names by Using the DATA Step

The following example uses the DATA step to change the order of first and last names.

```sas
/* Create a data set that contains a list of names. */
data ReversedNames;
  input name & $32.;
datalines;
  Jones, Fred
  Kavich, Kate
  Turley, Ron
  Dulix, Yolanda
;
/* Reverse last and first names with a DATA step. */
options pageno=1 nodate ls=80 ps=64;
data names;
  set ReversedNames;
  name = prxchange(s/\w+, (\w+)/$2 $1/, -1, name);
run;

proc print data=names;
run;
```

Output 4.34 Output from the DATA Step

<table>
<thead>
<tr>
<th>Obs</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Fred Jones</td>
</tr>
<tr>
<td>2</td>
<td>Kate Kavich</td>
</tr>
<tr>
<td>3</td>
<td>Ron Turley</td>
</tr>
<tr>
<td>4</td>
<td>Yolanda Dulix</td>
</tr>
</tbody>
</table>

Changing the Order of First and Last Names by Using PROC SQL

The following example uses PROC SQL to change the order of first and last names.

```sas
data ReversedNames;
  input name & $32.;
datalines;
  Jones, Fred
  Kavich, Kate
  Turley, Ron
  Dulix, Yolanda
;
```
proc sql;
  create table names as
  select prxchange('s/(\w+), (\w+)/$2 $1/', -1, name) as name
  from ReversedNames;
quit;

options pageno=1 nodate ls=80 ps=64;
proc print data=names;
run;

Output 4.35  Output from PROC SQL

<table>
<thead>
<tr>
<th>Obs</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Fred Jones</td>
</tr>
<tr>
<td>2</td>
<td>Kate Kavich</td>
</tr>
<tr>
<td>3</td>
<td>Ron Turley</td>
</tr>
<tr>
<td>4</td>
<td>Yolanda Dulix</td>
</tr>
</tbody>
</table>

Example 2: Matching Rows That Have the Same Name  The following example compares the names in two data sets, and writes those names that are common to both data sets.

data names;
  input name $32.;
datalines;
Ron Turley
Judy Donnelly
Kate Kavich
Tully Sanchez
;

data ReversedNames;
  input name $32.;
datalines;
Jones, Fred
Kavich, Kate
Turley, Ron
Dulix, Yolanda
;

options pageno=1 nodate ls=80 ps=64;
proc sql;
  create table NewNames as
  select a.name from names as a, ReversedNames as b
  where a.name = prxchange('s/(\w+), (\w+)/$2 $1/', -1, b.name);
quit;

proc print data=NewNames;
run;
Output 4.36  Output from Matching Rows That Have the Same Names

<table>
<thead>
<tr>
<th>The SAS System</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obs</td>
<td>name</td>
</tr>
<tr>
<td>1</td>
<td>Ron Turley</td>
</tr>
<tr>
<td>2</td>
<td>Kate Kavich</td>
</tr>
</tbody>
</table>

See Also

Functions and CALL routines:
“CALL PRXCHANGE Routine” on page 353
“CALL PRXDEBUG Routine” on page 355
“CALL PRXFREE Routine” on page 358
“CALL PRXNEXT Routine” on page 359
“CALL PRXPOSN Routine” on page 361
“CALL PRXSUBSTR Routine” on page 363
“PRXMATCH Function” on page 743
“PRXPAREN Function” on page 747
“PRXPARSE Function” on page 749
“PRXPOSN Function” on page 751

PRXMATCH Function

Searches for a pattern match and returns the position at which the pattern is found

Category: Character String Matching

Syntax

PRXMATCH (regular-expression-id | perl-regular-expression, source)

Arguments

position
specifies the numeric position in source at which the pattern begins. If no match is found, PRXMATCH returns a zero.

regular-expression-id
specifies a numeric pattern identifier that is returned by the PRXPARSE function.

Restriction: If you use this argument, you must also use the PRXPARSE function.

perl-regular-expression
specifies a Perl regular expression.
source

specifies the character expression that you want to search.

Details

The Basics  If you use regular-expression-id, then the PRXMATCH function searches source with the regular-expression-id that is returned by PRXPARSE, and returns the position at which the string begins. If there is no match, PRXMATCH returns a zero.

If you use perl-regular-expression, PRXMATCH searches source with the perl-regular-expression, and you do not need to call PRXPARSE.

You can use PRXMATCH with a Perl regular expression in a WHERE clause and in PROC SQL. For more information about pattern matching, see “Pattern Matching Using SAS Regular Expressions (RX) and Perl Regular Expressions (PRX)” on page 260.

Compiling a Perl Regular Expression  If perl-regular-expression is a constant or if it uses the /o option, then the Perl regular expression is compiled once and each use of PRXMATCH reuses the compiled expression. If perl-regular-expression is not a constant and if it does not use the /o option, then the Perl regular expression is recompiled for each call to PRXMATCH.

Note: The compile-once behavior occurs when you use PRXMATCH in a DATA step, in a WHERE clause, or in PROC SQL. For all other uses, the perl-regular-expression is recompiled for each call to PRXMATCH.

Comparisons

The Perl regular expression (PRX) functions and CALL routines work together to manipulate strings that match patterns. To see a list and short description of these functions and CALL routines, see the Character String Matching category in “Functions and CALL Routines by Category” on page 270.

Examples

Example 1: Finding the Position of a Substring in a String

Finding the Position of a Substring by Using PRXPARSE  The following example searches a string for a substring, and returns its position in the string.

```sas
data _null_
  /* Use PRXPARSE to compile the Perl regular expression. */
  patternID = prxparse('/world/');
  /* Use PRXMATCH to find the position of the pattern match. */
  position=prxmatch(patternID, 'Hello world!');
  put position=;
run;

SAS writes the following line to the log:

position=7
```

Finding the Position of a Substring by Using a Perl Regular Expression  The following example uses a Perl regular expression to search a string (Hello world) for a substring (world) and to return the position of the substring in the string.

```sas
data _null_
  /* Use PRXPARSE to compile the Perl regular expression. */
```
position=prxmatch('/world/', 'Hello world!');
put position=;
run;

SAS writes the following line to the SAS:
position=7

Example 2: Finding the Position of a Substring in a String: A Complex Example

The following example uses several Perl regular expression functions and a CALL routine to find the position of a substring in a string.

data _null_
if _N_ = 1 then
  do;
    retain PerlExpression;
    pattern = "/(\d+):(\d\d)(?:\.\d+)?/";
    PerlExpression = prxparse(pattern);
  end;

  array match[3] $ 8;
  input minsec $80.;
  position = prxmatch(PerlExpression, minsec);
  if position ^= 0 then
    do;
      do i = 1 to prxparen(PerlExpression);
        call prxposn(PerlExpression, i, start, length);
        if start ^= 0 then
          match[i] = substr(minsec, start, length);
        end;
      if ^missing(match[3]) then
        put ", " match[3] "milliseconds";
    end;
  datalines;
14:56.456
45:32
;
run;

The following lines are written to the SAS log:
14 minutes, 56 seconds, 456 milliseconds
45 minutes, 32 seconds

Example 3: Extracting a Zip Code from a Data Set

Extracting a Zip Code by Using the DATA Step

The following example uses a DATA step to search each observation in a data set for a nine-digit zip code, and writes those observations to the data set ZipPlus4.

data ZipCodes;
  input name: $16. zip:$10.;
datalines;
/* Extract ZIP+4 ZIP codes with the DATA step. */
data ZipPlus4;
  set ZipCodes;
  where prxmatch('/\d{5}-\d{4}/', zip);
run;

options nodate pageno=1 ls=80 ps=64;
proc print data=ZipPlus4;
run;

Output 4.37  Zip Code Output from the DATA Step

<table>
<thead>
<tr>
<th>Obs</th>
<th>Name</th>
<th>Zip Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Johnathan</td>
<td>32523-2343</td>
</tr>
<tr>
<td>2</td>
<td>Samuel</td>
<td>93849-3843</td>
</tr>
</tbody>
</table>

Extracting a Zip Code by Using PROC SQL  The following example searches each observation in a data set for a nine-digit zip code, and writes those observations to the data set ZipPlus4.

data ZipCodes;
  input name: $16. zip:$10.;
datalines;
Johnathan 32523-2343
Seth 85030
Kim 39204
Samuel 93849-3843
;

/* Extract ZIP+4 ZIP codes with PROC SQL. */
proc sql;
  create table ZipPlus4 as
  select * from ZipCodes
  where prxmatch('/\d{5}-\d{4}/', zip);
run;

options nodate pageno=1 ls=80 ps=64;
proc print data=ZipPlus4;
run;
### PRXPAREN Function

**Returns the last bracket match for which there is a match in a pattern**

**Category:** Character String Matching

**Restriction:** Use with the PR XPARSE function

#### Syntax

\[
\text{PRXPAREN}(\text{regular-expression-id})
\]

#### Arguments

**regular-expression-id**

specifies a numeric identification number that is returned by the PR XPARSE function.

#### Details

The PRXPAREN function is useful in finding the largest capture-buffer number that can be passed to the CALL PRXPOSN routine, or in identifying which part of a pattern matched.
For more information about pattern matching, see “Pattern Matching Using SAS Regular Expressions (RX) and Perl Regular Expressions (PRX)” on page 260.

**Comparisons**

The Perl regular expression (PRX) functions and CALL routines work together to manipulate strings that match patterns. To see a list and short description of these functions and CALL routines, see the Character String Matching category in “Functions and CALL Routines by Category” on page 270.

**Examples**

The following example uses Perl regular expressions and writes the results to the SAS log.

```sas
data _null_
  ExpressionID = prxparse('/(magazine)|(book)|(newspaper)/');
  position = prxmatch(ExpressionID, 'find book here');
  if position then paren = prxparen(ExpressionID);
  put 'Matched paren ' paren;

  position = prxmatch(ExpressionID, 'find magazine here');
  if position then paren = prxparen(ExpressionID);
  put 'Matched paren ' paren;

  position = prxmatch(ExpressionID, 'find newspaper here');
  if position then paren = prxparen(ExpressionID);
  put 'Matched paren ' paren;
run;
```

The following lines are written to the SAS log:

- Matched paren 2
- Matched paren 1
- Matched paren 3

**See Also**

Functions and CALL routines:

- “CALL PRXCHANGE Routine” on page 353
- “CALL PRXDEBUG Routine” on page 355
- “CALL PRXFREE Routine” on page 358
- “CALL PRXNEXT Routine” on page 359
- “CALL PRXPOSN Routine” on page 361
- “CALL PRXSUBSTR Routine” on page 363
- “CALL PRXCHANGE Routine” on page 353
- “PRXCHANGE Function” on page 739
- “PRXMATCH Function” on page 743
- “PRXPARSE Function” on page 749
- “PRXPOSN Function” on page 751
**PRXPARSE Function**

Compiles a Perl regular expression (PRX) that can be used for pattern matching of a character value

**Category:** Character String Matching  
**Restriction:** Use with other Perl regular expressions.

**Syntax**

```
regular-expression-id=PRXPARSE (perl-regular-expression)
```

**Arguments**

- `regular-expression-id` specifies a numeric pattern identifier that is returned by the PRXPARSE function.
- `perl-regular-expression` specifies a character value that is a Perl regular expression.

**Details**

**The Basics**  
The PRXPARSE function returns a pattern identifier number that is used by other Perl functions and CALL routines to match patterns. If an error occurs in parsing the regular expression, SAS returns a missing value.

PRXPARSE uses metacharacters in constructing a Perl regular expression. To view a table of common metacharacters, see “Syntax of Perl Regular Expressions” on page 262.

For more information about pattern matching, see “Pattern Matching Using SAS Regular Expressions (RX) and Perl Regular Expressions (PRX)” on page 260.

**Compiling a Perl Regular Expression**  
If `perl-regular-expression` is a constant or if it uses the /o option, the Perl regular expression is compiled only once. Successive calls to PRXPARSE will not cause a recompile, but will return the `regular-expression-id` for the regular expression that was already compiled. This behavior simplifies the code because you do not need to use an initialization block (IF _N_ =1) to initialize Perl regular expressions.

*Note:* If you have a Perl regular expression that is a constant, or if the regular expression uses the /o option, then calling PRXFREE to free the memory allocation results in the need to recompile the regular expression the next time that it is called by PRXPARSE.

The compile-once behavior occurs when you use PRXPARSE in a DATA step. For all other uses, the `perl-regular-expression` is recompiled for each call to PRXPARSE.

**Comparisons**

The Perl regular expression (PRX) functions and CALL routines work together to manipulate strings that match patterns. To see a list and short description of these functions and CALL routines, see the Character String Matching category in “Functions and CALL Routines by Category” on page 270.
Examples

The following example uses metacharacters and regular characters to construct a Perl regular expression. The example parses addresses and writes formatted results to the SAS log.

```sas
data _null_
  if _N_ = 1 then
    do;
      retain patternID;

      /* The i option specifies a case insensitive search. */
      pattern = "/ave|avenue|dr|drive|rd|road/i";
      patternID = prxparse(pattern);
    end;

  input street $80.;
  call prxsubstr(patternID, street, position, length);
  if position ^= 0 then
    do;
      match = substr(street, position, length);
      put match:$QUOTE. "found in " street:$QUOTE.;
    end;

datalines;
153 First Street
6789 64th Ave
4 Moritz Road
7493 Wilkes Place
;

The following lines are written to the SAS log:

"Ave" found in "6789 64th Ave"
"Road" found in "4 Moritz Road"
```

See Also

Functions and CALL routines:

“CALL PRXCHANGE Routine” on page 353
“CALL PRXDEBUG Routine” on page 355
“CALL PRXFREE Routine” on page 358
“CALL PRXNEXT Routine” on page 359
“CALL PRXPOSN Routine” on page 361
“CALL PRXSUBSTR Routine” on page 363
“CALL PRXCHANGE Routine” on page 353
“PRXCHANGE Function” on page 739
“PRXPARSE Function” on page 747
“PRXMATCH Function” on page 743
“PRXPOSN Function” on page 751
PRXPOSN Function

Returns the value for a capture buffer

Category: Character String Matching
Restriction: Use with the PRXPARSE function

Syntax

PRXPOSN(regular-expression-id, capture-buffer, source)

Arguments

regular-expression-id
specifies a numeric pattern identifier that is returned by the PRXPARSE function.

capture-buffer
is a numeric value that identifies the capture buffer for which to retrieve a value:

  □ If the value of capture-buffer is zero, PRXPOSN returns the entire match.
  □ If the value of capture-buffer is between 1 and the number of open parentheses
     in the regular expression, then PRXPOSN returns the value for that capture
     buffer.
  □ If the value of capture-buffer is greater than the number of open parentheses,
     then PRXPOSN returns a missing value.

source
specifies the text from which to extract capture buffers.

Details

The PRXPOSN function uses the results of PRXMATCH, PRXSUBSTR, PRXCHANGE,
or PRXNEXT to return a capture buffer. A match must be found by one of these
functions for PRXPOSN to return meaningful information.

A capture buffer is part of a match, enclosed in parentheses, that is specified in a
regular expression. This function simplifies using capture buffers by returning the text
for the capture buffer directly, and by not requiring a call to SUBSTR as in the case of
CALL PRXPOSN.

For more information about pattern matching, see “Pattern Matching Using SAS
Regular Expressions (RX) and Perl Regular Expressions (PRX)” on page 260.

Comparisons

The PRXPOSN function is similar to the CALL PRXPOSN routine, except that it returns
the capture buffer itself rather than the position and length of the capture buffer.

The Perl regular expression (PRX) functions and CALL routines work together to
manipulate strings that match patterns. To see a list and short description of these
functions and CALL routines, see the Character String Matching category in “Functions
and CALL Routines by Category” on page 270.
Examples

Example 1: Extracting First and Last Names

The following example uses PRXPOSN to extract first and last names from a data set.

```sas
data ReversedNames;
  input name & $32.;
datalines;
Jones, Fred
Kavich, Kate
Turley, Ron
Dulix, Yolanda
;

data FirstLastNames;
  length first last $ 16;
  keep first last;
  retain re;
  if _N_ = 1 then
    re = prxparse('/\w+, \w+/');
  set ReversedNames;
  if prxmatch(re, name) then
    do;
      last = prxposn(re, 1, name);
      first = prxposn(re, 2, name);
    end;
  run;

options pageno=1 nodate ls=80 ps=64;
proc print data = FirstLastNames;
run;
```

Output 4.39  Output from PRXPOSN: First and Last Names

<table>
<thead>
<tr>
<th>Obs</th>
<th>first</th>
<th>last</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Fred</td>
<td>Jones</td>
</tr>
<tr>
<td>2</td>
<td>Kate</td>
<td>Kavich</td>
</tr>
<tr>
<td>3</td>
<td>Ron</td>
<td>Turley</td>
</tr>
<tr>
<td>4</td>
<td>Yolanda</td>
<td>Dulix</td>
</tr>
</tbody>
</table>

Example 2: Extracting Names When Some Names Are Invalid

The following example creates a data set that contains a list of names. Observations that have only a first name or only a last name are invalid. PRXPOSN extracts the valid names from the data set, and writes the names to the data set NEW.

```sas
data old;
  input name $60.;
datalines;
Judith S Reaveley
Ralph F. Morgan
Jess Ennis
;
```
Carol Echols
Kelly Hansen Huff
Judith
Nick
Jones

; data new;
  length first middle last $ 40;
  keep first middle last;
  re = prxparse('/(\S+)\s+(\^[s]s+)?(\S+)/o');
  set old;
  if prxmatch(re, name) then
    do;
      first = prxposn(re, 1, name);
      middle = prxposn(re, 2, name);
      last = prxposn(re, 3, name);
      output;
    end;
  run;

options pageno=1 nodate ls=80 ps=64;
proc print data = new;
run;

Output 4.40  Output of Valid Names

<table>
<thead>
<tr>
<th>Obs</th>
<th>first</th>
<th>middle</th>
<th>last</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Judith</td>
<td>S</td>
<td>Reaveley</td>
</tr>
<tr>
<td>2</td>
<td>Ralph</td>
<td>F.</td>
<td>Morgan</td>
</tr>
<tr>
<td>3</td>
<td>Jess</td>
<td>Ennis</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Carol</td>
<td>Echols</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Kelly</td>
<td>Hansen</td>
<td>Huff</td>
</tr>
</tbody>
</table>

See Also

Functions:

“CALL PRXCHANGE Routine” on page 353
“CALL PRXDEBUG Routine” on page 355
“CALL PRXFREE Routine” on page 358
“CALL PRXNEXT Routine” on page 359
“CALL PRXPOSN Routine” on page 361
“CALL PRXSUBSTR Routine” on page 363
“CALL PRXCHANGE Routine” on page 353
“PRXCHANGE Function” on page 739
“PRXMATCH Function” on page 743
PTRLONGADD Function

Returns the pointer address as a character variable on 32-bit and 64-bit platforms

Category: Special

Syntax

PTRADDLONG(pointer<,amount>)

Arguments

pointer
specifies a character string that is the pointer address.

amount
specifies the amount to add to the address.

Tip: amount can be a negative number.

Details

The PTRLONGADD function performs pointer arithmetic and returns a pointer address as a character string.

Examples

The following example returns the pointer address for the variable Z.

data _null_
  x='ABCDE'
  y=ptrlongadd(addrlong(x),2)
  z=peekclong(y,1)
  put z=
run;

The output from the SAS log is: z=C

PUT Function

Returns a value using a specified format

Category: Special
Syntax

PUT(source, format.)

Arguments

source
identifies the SAS variable or constant whose value you want to reformat. The source argument can be character or numeric.

format.
contains the SAS format that you want applied to the variable or constant that is specified in the source. 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.

Restriction: The format must be of the same type as the source, either character or numeric.

Details

If the PUT function returns a value to a variable that has not yet been assigned a length, by default the variable length is determined by the width of the format.

The format must be the same type (numeric or character) as the value of source. The result of the PUT function is always a character string. If the source is numeric, the resulting string is right aligned. If the source is character, the result is left aligned.

Use PUT to convert a numeric value to a character value. PUT writes (or produces a reformatted result) only while it is executing. To preserve the result, assign it to a variable.

Comparisons

The PUT statement and the PUT function are similar. The PUT function returns a value using a specified format. You must use an assignment statement to store the value in a variable. The PUT statement writes a value to an external destination (either the SAS log or a destination you specify).

Examples

Example 1: Converting Numeric Values to Character Value
In this example, the first statement converts the values of CC, a numeric variable, into the four-character hexadecimal format, and the second writes the same value that the PUT function returns.

```sas
cchex=put(cc,hex4.);
put cc hex4.;
```

Example 2: Using PUT and INPUT Functions
In this example, the PUT function returns a numeric value as a character string. The value 122591 is assigned to the CHARDATE variable. The INPUT function returns the value of the character string as a SAS date value using a SAS date informat. The value 11681 is stored in the SASDATE variable.
numdate=122591;
chardate=put(numdate,z6.);
sasdate=input(chardate,mmddyy6.);

See Also

Functions:
  “INPUT Function” on page 592
  “INPUTC Function” on page 594
  “INPUTN Function” on page 596
  “PUTC Function” on page 756,
  “PUTN Function” on page 758

Statement:
  “PUT Statement” on page 1344

---

PUTC Function

Enables you to specify a character format at run time

Category:   Special

Syntax

PUTC(source, format.<,w>)

Arguments

source
  is the SAS expression to which you want to apply the format.

format.
  is an expression that contains the character format you want to apply to source.

w
  specifies a width to apply to the format.

Interaction:   If you specify a width here, it overrides any width specification in the
  format.

Details

If the PUTC function returns a value to a variable that has not yet been assigned a
length, by default the variable length is determined by the length of the first argument.

Comparisons

The PUTN function enables you to specify a numeric format at run time.
Examples

Example 1: Specifying a Character Format  The PROC FORMAT step in this example creates a format, TYPEFMT, that formats the variable values 1, 2, and 3 with the name of one of the three other formats that this step creates. These three formats output responses of "positive," "negative," and "neutral" as different words, depending on the type of question. After PROC FORMAT creates the formats, the DATA step creates a SAS data set from raw data consisting of a number identifying the type of question and a response. After reading a record, the DATA step uses the value of TYPE to create a variable, RESPFMT, that contains the value of the appropriate format for the current type of question. The DATA step also creates another variable, WORD, whose value is the appropriate word for a response. The PUTC function assigns the value of WORD based on the type of question and the appropriate format.

```sparql
proc format;
  value typefmt 1='$groupx'
    2='$groupy'
    3='$groupz';
  value $groupx 'positive'='agree'
    'negative'='disagree'
    'neutral'='notsure';
  value $groupy 'positive'='accept'
    'negative'='reject'
    'neutral'='possible';
  value $groupz 'positive'='pass'
    'negative'='fail'
    'neutral'='retest';
run;

data answers;
  length word $ 8;
  input type response $;
  respfmt = put(type, typefmt.);
  word = putc(response, respfmt);
datalines;
  1 positive
  1 negative
  1 neutral
  2 positive
  2 negative
  2 neutral
  3 positive
  3 negative
  3 neutral
;
```

The value of the variable WORD is agree for the first observation. The value of the variable WORD is retest for the last observation.

See Also

Functions:

“INPUT Function” on page 592
PUTN Function

Enables you to specify a numeric format at run time

Category: Special

Syntax

PUTN(source, format.<,w,<d>>)  

Arguments

source
    is the SAS expression to which you want to apply the format.

format.
    is an expression that contains the numeric format you want to apply to source.

w
    specifies a width to apply to the format.
    Interaction: If you specify a width here, it overrides any width specification in the format.

d
    specifies the number of decimal places to use.
    Interaction: If you specify a number here, it overrides any decimal-place specification in the format.

Details

If the PUTN function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

Comparisons

The PUTC function enables you to specify a character format at run time.

Examples

Example 1: Specifying a Numeric Format  The PROC FORMAT step in this example creates a format, WRITFMT., that formats the variable values 1 and 2 with the name of a SAS date format. The DATA step creates a SAS data set from raw data consisting of a number and a key. After reading a record, the DATA step uses the value of KEY to create a variable, DATEFMT, that contains the value of the appropriate date format.
The DATA step also creates a new variable, DATE, whose value is the formatted value of the date. PUTN assigns the value of DATE based on the value of NUMBER and the appropriate format.

```plaintext
proc format;
  value writfmt 1='date9.'
                  2='mmddyy10.';
run;
data dates;
  input number key;
  datefmt=put(key,writfmt.);
  date=putn(number,datefmt);
datalines;
  15756 1
  14552 2
;
```

See Also

Functions:

- “INPUT Function” on page 592
- “INPUTC Function” on page 594
- “INPUTN Function” on page 596
- “PUT Function” on page 754
- “PUTC Function” on page 756

---

**PVP Function**

Returns the present value for a periodic cash flow stream with repayment of principal at maturity, such as a bond

**Category:** Financial

**Syntax**

\[ \text{PVP}(A,c,n,K,k_0,y) \]

**Arguments**

- \( A \)
  - Specifies the par value.
  - **Range:** \( A > 0 \)

- \( c \)
  - Specifies the nominal per-year coupon rate, expressed as a fraction.
  - **Range:** \( 0 \leq c < 1 \)
\( n \)
specifies the number of coupons per year.
**Range:** \( n > 0 \) and is an integer

\( K \)
specifies the number of remaining coupons.
**Range:** \( K > 0 \) and is an integer

\( k_0 \)
specifies the time from the present date to the first coupon date, expressed in terms of the number of years.
**Range:** \( 0 < k_0 \leq \frac{1}{n} \)

\( y \)
specifies the nominal per-year yield-to-maturity, expressed as a fraction.
**Range:** \( y > 0 \)

**Details**
The PVP function is based on the relationship

\[
P = \sum_{k=1}^{K} \frac{c(k)}{(1 + \frac{y}{n})^{t_k}}
\]

where
\[
t_k = n k_0 + k - 1
\]
\[
c(k) = \frac{c}{n} A \quad \text{for } k = 1, \ldots, K - 1
\]
\[
c(K) = (1 + \frac{c}{n}) A
\]

**Examples**

data _null_;  
p=pvp(1000,.01,4,14,.33/2,.10);  
put p;  
run;  
The value returned is 743.168.

**QTR Function**

Returns the quarter of the year from a SAS date value

**Category:** Date and Time

**Syntax**

\[ \text{QTR}(\text{date}) \]
Arguments

date
specifies a SAS expression that represents a SAS date value.

Details
The QTR function returns a value of 1, 2, 3, or 4 from a SAS date value to indicate the quarter of the year in which a date value falls.

Examples
The following SAS statements produce these results.

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x='20jan94'd;</td>
<td></td>
</tr>
<tr>
<td>y=qtr(x);</td>
<td></td>
</tr>
<tr>
<td>put y=;</td>
<td>y=1</td>
</tr>
</tbody>
</table>

See Also
Function:
“YYQ Function” on page 920

QUANTILE Function
Computes the quantile from a specified distribution
Category: Quantile
See: “CDF Function” on page 416

Syntax
QUANTILE(‘dist’, probability, parm-1,...,parm-k)

Arguments
'dist'
is a character string that identifies the distribution. Valid distributions are as follows:

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bernoulli</td>
<td>‘BERNOULLI’</td>
</tr>
<tr>
<td>Beta</td>
<td>‘BETA’</td>
</tr>
<tr>
<td>Binomial</td>
<td>‘BINOMIAL’</td>
</tr>
</tbody>
</table>
### Distribution Argument

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cauchy</td>
<td>‘CAUCHY’</td>
</tr>
<tr>
<td>Chi-Square</td>
<td>‘CHISQUARE’</td>
</tr>
<tr>
<td>Exponential</td>
<td>‘EXPONENTIAL’</td>
</tr>
<tr>
<td>F</td>
<td>‘F’</td>
</tr>
<tr>
<td>Gamma</td>
<td>‘GAMMA’</td>
</tr>
<tr>
<td>Geometric</td>
<td>‘GEOMETRIC’</td>
</tr>
<tr>
<td>Hypergeometric</td>
<td>‘HYPERGEOMETRIC’</td>
</tr>
<tr>
<td>Laplace</td>
<td>‘LAPLACE’</td>
</tr>
<tr>
<td>Logistic</td>
<td>‘LOGISTIC’</td>
</tr>
<tr>
<td>Lognormal</td>
<td>‘LOGNORMAL’</td>
</tr>
<tr>
<td>Negative binomial</td>
<td>‘NEGBINOMIAL’</td>
</tr>
<tr>
<td>Normal</td>
<td>‘NORMAL’</td>
</tr>
<tr>
<td>Normal mixture</td>
<td>‘NORMALMIX’</td>
</tr>
<tr>
<td>Pareto</td>
<td>‘PARETO’</td>
</tr>
<tr>
<td>Poisson</td>
<td>‘POISSON’</td>
</tr>
<tr>
<td>T</td>
<td>‘T’</td>
</tr>
<tr>
<td>Uniform</td>
<td>‘UNIFORM’</td>
</tr>
<tr>
<td>Wald (inverse Gaussian)</td>
<td>‘WALD’</td>
</tr>
<tr>
<td>Weibull</td>
<td>‘WEIBULL’</td>
</tr>
</tbody>
</table>

**Note:** Except for T, F, and NORMALMIX, you can minimally identify any distribution by its first four characters.

#### probability

is a numeric random variable.

#### parm-1,...,parm-k

are optional shape, location, or scale parameters appropriate for the specific distribution.

The QUANTILE function computes the probability from various continuous and discrete distributions. For more information, see the on page 417.

### Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>y=quantile(‘BERN’,.75,.25);</td>
<td>0</td>
</tr>
<tr>
<td>y=quantile(‘BETA’,0.1,3,4);</td>
<td>0.2009088789</td>
</tr>
<tr>
<td>y=quantile(‘BINOM’,4,.5,10);</td>
<td>5</td>
</tr>
<tr>
<td>y=quantile(‘CAUCHY’,.85);</td>
<td>1.9626105055</td>
</tr>
</tbody>
</table>
QUOTE Function

Adds double quotation marks to a character value

Category: Character

Syntax

QUOTE(argument)

Arguments

*argument* is a character value.

Details

If the QUOTE function returns a value to a variable that has not yet been assigned a length, then by default the variable is assigned a length of 200.

The QUOTE function adds double quotation marks, the default character, to a character value. If double quotation marks are found within the argument, they are doubled in the output.

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>y=quantile('CHISQ',.6,11);</code></td>
<td>11.529833841</td>
</tr>
<tr>
<td><code>y=quantile('EXPO',.6);</code></td>
<td>0.9162907319</td>
</tr>
<tr>
<td><code>y=quantile('F',8,2,3);</code></td>
<td>2.8860266073</td>
</tr>
<tr>
<td><code>y=quantile('GAMMA',.4,3);</code></td>
<td>2.285076904</td>
</tr>
<tr>
<td><code>y=quantile('HYPER',.5,200,50,10);</code></td>
<td>2</td>
</tr>
<tr>
<td><code>y=quantile('LAPLACE',.8);</code></td>
<td>0.9162907319</td>
</tr>
<tr>
<td><code>y=quantile('LOGISTIC',.7);</code></td>
<td>0.8472978604</td>
</tr>
<tr>
<td><code>y=quantile('LOGNORMAL',.5);</code></td>
<td>1</td>
</tr>
<tr>
<td><code>y=quantile('NEGB',.5,.5,2);</code></td>
<td>1</td>
</tr>
<tr>
<td><code>y=quantile('NORMAL',.975);</code></td>
<td>1.9599639845</td>
</tr>
<tr>
<td><code>y=quantile('PARETO',.01,1);</code></td>
<td>1.0101010101</td>
</tr>
<tr>
<td><code>y=quantile('POISSON',.9,1);</code></td>
<td>2</td>
</tr>
<tr>
<td><code>y=quantile('T',.8,5);</code></td>
<td>0.9195437802</td>
</tr>
<tr>
<td><code>y=quantile('UNIFORM',0.25);</code></td>
<td>0.25</td>
</tr>
<tr>
<td><code>y=quantile('WALD',.6,2);</code></td>
<td>0.9526209927</td>
</tr>
<tr>
<td><code>y=quantile('WEIBULL',.6,2);</code></td>
<td>0.9572307621</td>
</tr>
</tbody>
</table>
The length of the receiving variable must be long enough to contain the argument (including trailing blanks), leading and trailing quotation marks, and any embedded quotation marks that are doubled. For example, if the argument is ABC followed by three trailing blanks, then the receiving variable must have a length of at least eight to hold “ABC###”. (The character # represents a blank space.) If the receiving field is not long enough, the QUOTE function returns a blank string, and writes an invalid argument note to the log.

**Examples**

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x='A&quot;B'</td>
<td></td>
</tr>
<tr>
<td>y=quote(x);</td>
<td>&quot;A&quot;&quot;B&quot;</td>
</tr>
<tr>
<td>put y;</td>
<td></td>
</tr>
<tr>
<td>x='A''B';</td>
<td></td>
</tr>
<tr>
<td>y=quote(x);</td>
<td>&quot;A'B&quot;</td>
</tr>
<tr>
<td>put y;</td>
<td></td>
</tr>
<tr>
<td>x='Paul''s';</td>
<td></td>
</tr>
<tr>
<td>y=quote(x);</td>
<td>&quot;Paul's&quot;</td>
</tr>
<tr>
<td>put y;</td>
<td></td>
</tr>
<tr>
<td>x='Paul''s Catering Service ';</td>
<td></td>
</tr>
<tr>
<td>y=quote(trim(x));</td>
<td>&quot;Paul's Catering Service&quot;</td>
</tr>
<tr>
<td>put y;</td>
<td></td>
</tr>
</tbody>
</table>

**RANBIN Function**

*Returns a random variate from a binomial distribution*

**Category:** Random Number

**Tip:** If you want to change the seed value during execution, you must use the CALL RANBIN routine instead of the RANBIN function.

**Syntax**

RANBIN(seed,n,p)

**Arguments**

*seed*

is an integer. If seed ≤ 0, the time of day is used to initialize the seed stream.

- **Range:** seed < 2^{31}−1
- **See:** “Seed Values” on page 257 for more information about seed values

*n*
is an integer number of independent Bernoulli trials parameter.

**Range:** \( n > 0 \)

\( p \)

is a numeric probability of success parameter.

**Range:** \( 0 < p < 1 \)

### Details

The RANBIN function returns a variate that is generated from a binomial distribution with mean \( np \) and variance \( np(1-p) \). If \( n \leq 50, np \leq 5, \) or \( n(1-p) \leq 5, \) an inverse transform method applied to a RANUNI uniform variate is used. If \( n > 50, np > 5, \) and \( n(1-p) > 5, \) the normal approximation to the binomial distribution is used. In that case, the Box-Muller transformation of RANUNI uniform variates is used.

### Comparisons

The CALL RANBIN routine, an alternative to the RANBIN function, gives greater control of the seed and random number streams.

### See Also

Call routine:

“CALL RANBIN Routine” on page 366

---

**RANCAU Function**

**Returns a random variate from a Cauchy distribution**

**Category:** Random Number

**Tip:** If you want to change the seed value during execution, you must use the CALL RANCAU routine instead of the RANCAU function.

### Syntax

RANCAU(seed)

### Arguments

**seed**

is an integer. If \( seed \leq 0, \) the time of day is used to initialize the seed stream.

**Range:** \( seed < 2^{31} - 1 \)

**See:** “Seed Values” on page 257 for more information about seed values

### Details

The RANCAU function returns a variate that is generated from a Cauchy distribution with location parameter 0 and scale parameter 1. An acceptance-rejection procedure...
applied to RANUNI uniform variates is used. If \( u \) and \( v \) are independent uniform \((-1/2, 1/2)\) variables and \( u^2 + v^2 \leq 1/4 \), then \( u/v \) is a Cauchy variate. A Cauchy variate \( X \) with location parameter \( \text{ALPHA} \) and scale parameter \( \text{BETA} \) can be generated:

\[
x = \text{ALPHA} + \text{BETA} \times \text{rancau} (\text{seed});
\]

**Comparisons**

The CALL RANCAU routine, an alternative to the RANCAU function, gives greater control of the seed and random number streams.

**See Also**

Call routine:

"CALL RANCAU Routine" on page 368

---

**RAND Function**

Generates random numbers from a specified distribution

**Category:** Random Number

**Syntax**

\[
\text{RAND} ('\text{dist}', \text{parm-1},...,\text{parm-k})
\]

**Arguments**

'\text{dist}'

is a character string that identifies the distribution. Valid distributions are as follows:

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bernoulli</td>
<td>'BERNOULLI'</td>
</tr>
<tr>
<td>Beta</td>
<td>'BETA'</td>
</tr>
<tr>
<td>Binomial</td>
<td>'BINOMIAL'</td>
</tr>
<tr>
<td>Cauchy</td>
<td>'CAUCHY'</td>
</tr>
<tr>
<td>Chi-Square</td>
<td>'CHISQUARE'</td>
</tr>
<tr>
<td>Erlang</td>
<td>'ERLANG'</td>
</tr>
<tr>
<td>Exponential</td>
<td>'EXPONENTIAL'</td>
</tr>
<tr>
<td>F</td>
<td>'F'</td>
</tr>
<tr>
<td>Gamma</td>
<td>'GAMMA'</td>
</tr>
<tr>
<td>Geometric</td>
<td>'GEOMETRIC'</td>
</tr>
</tbody>
</table>
### Functions and CALL Routines

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hypergeometric</td>
<td>'HYPERGEOMETRIC'</td>
</tr>
<tr>
<td>Lognormal</td>
<td>'LOGNORMAL'</td>
</tr>
<tr>
<td>Negative binomial</td>
<td>'NEGBINOMIAL'</td>
</tr>
<tr>
<td>Normal</td>
<td>'NORMAL'</td>
</tr>
<tr>
<td>Poisson</td>
<td>'POISSON'</td>
</tr>
<tr>
<td>T</td>
<td>'T'</td>
</tr>
<tr>
<td>Tabled</td>
<td>'TABLE'</td>
</tr>
<tr>
<td>Triangular</td>
<td>'TRIANGLE'</td>
</tr>
<tr>
<td>Uniform</td>
<td>'UNIFORM'</td>
</tr>
<tr>
<td>Weibull</td>
<td>'WEIBULL'</td>
</tr>
</tbody>
</table>

**Note:** Except for T and F, you can minimally identify any distribution by its first four characters.

**parm-1,...,parm-k**

are *shape*, *location*, or *scale* parameters appropriate for the specific distribution.

See: “Details” on page 767 for complete information about these parameters

### Details

The RAND function generates random numbers from various continuous and discrete distributions. Wherever possible, the simplest form of the distribution is used.

#### Reproducing a Random Number Stream

If you want to create reproducible streams of random numbers, then use the CALL STREAMINT routine to specify a seed value for random number generation. Do this once per DATA step before any invocation of the RAND function. If you omit the call to the STREAMINIT routine (or if you specify a non-positive seed value in the CALL STREAMINIT routine), then RAND uses a call to the system clock to seed itself. For more information, see CALL STREAMINT, Example 1 on page 400.

#### Bernoulli Distribution

\[ x = \text{RAND('BERNOULLI',p)} \]

where

\[ x \]

is an observation from the distribution with the following probability density function:

\[
f (x) = \begin{cases} 
  1 & \quad p = 0, x = 0 \\
  \frac{p^x}{1 - p} & \quad 0 < p < 1, x = 0, 1 \\
  1 & \quad p = 1, x = 1 
\end{cases}
\]

**Range:** \( x = 0, 1 \)
\( p \)

is a numeric probability of success.

Range: \( 0 \leq p \leq 1 \)

**Beta Distribution**

\( x = \text{RAND('BETA',}a,b) \)

where

\( x \)

is an observation from the distribution with the following probability density function:

\[
f(x) = \frac{\Gamma(a + b)}{\Gamma(a) \Gamma(b)} x^{a-1} (1 - x)^{b-1}
\]

Range: \( 0 < x < 1 \)

\( a \)

is a numeric shape parameter.

Range: \( a > 0 \)

\( b \)

is a numeric shape parameter.

Range: \( b > 0 \)

**Binomial Distribution**

\( x = \text{RAND('BINOMIAL',}p,n) \)

where

\( x \)

is an integer observation from the distribution with the following probability density function:

\[
f(x) = \begin{cases} 
\binom{n}{x} p^x (1 - p)^{n-x} & p = 0, x = 0 \\
\frac{1}{x!} & 0 < p < 1, x = 0, ..., n \\
p = 1, x = n
\end{cases}
\]

Range: \( x = 0, 1, ..., n \)

\( p \)

is a numeric probability of success.

Range: \( 0 \leq p \leq 1 \)

\( n \)

is an integer parameter that counts the number of independent Bernoulli trials.

Range: \( n = 1, 2, ... \)

**Cauchy Distribution**

\( x = \text{RAND('CAUCHY')} \)

where

\( x \)
is an observation from the distribution with the following probability density function:

$$f (x) = \frac{1}{\pi (1 + x^2)}$$

**Range:** $-\infty < x < \infty$

**Chi-Square Distribution**

\(x = \text{RAND('CHISQUARE', df)}\)

where

\(x\) is an observation from the distribution with the following probability density function:

$$f (x) = \frac{2^{-\frac{df}{2}} x^{\frac{df}{2}-1} e^{-\frac{x}{2}}}{\Gamma\left(\frac{df}{2}\right)}$$

**Range:** \(x > 0\)

\(df\) is a numeric degrees of freedom parameter.

**Range:** \(df > 0\)

**Erlang Distribution**

\(x = \text{RAND('ERLANG', a)}\)

where

\(x\) is an observation from the distribution with the following probability density function:

$$f (x) = \frac{1}{\Gamma(a)} x^{a-1} e^{-x}$$

**Range:** \(x > 0\)

\(a\) is an integer numeric shape parameter.

**Range:** \(a = 1, 2, \ldots\)
**Exponential Distribution**

\[ x = \text{RAND}('\text{EXPONENTIAL}') \]

where

\( x \)

is an observation from the distribution with the following probability density function:

\[ f(x) = e^{-x} \]

**Range:** \( x > 0 \)

**F Distribution**

\[ x = \text{RAND}('\text{F}', ndf, ddf) \]

where

\( x \)

is an observation from the distribution with the following probability density function:

\[
\frac{\Gamma \left( \frac{ndf+ddf}{2} \right) \, ndf^{ndf/2} \, ddf^{ddf/2} \, x^{ndf/2-1}}{\Gamma \left( \frac{ndf}{2} \right) \Gamma \left( \frac{ddf}{2} \right) (ddf + ndf x)^{(ndf+ddf)/2}}
\]

**Range:** \( x > 0 \)

\( ndf \)

is a numeric numerator degrees of freedom parameter.

**Range:** \( ndf > 0 \)

\( ddf \)

is a numeric denominator degrees of freedom parameter.

**Range:** \( ddf > 0 \)

**Gamma Distribution**

\[ x = \text{RAND}('\text{GAMMA}', a) \]

where

\( x \)

is an observation from the distribution with the following probability density function:

\[ f(x) = \frac{1}{\Gamma(a)} x^{a-1} e^{-x} \]

**Range:** \( x > 0 \)

\( a \)

is a numeric shape parameter.
Range: \( a > 0 \)

**Geometric Distribution**

\[
x = \text{RAND('GEOMETRIC',}p\text{'})
\]

where

\( x \)

is an integer observation from the distribution with the following probability density function:

\[
f(x) = \begin{cases} 
(1 - p)^{x-1} p & 0 < p < 1, x = 1, 2, ... \\
1 & p = 1, x = 1
\end{cases}
\]

Range: \( x = 1, 2, ... \)

\( p \)

is a numeric probability of success.

Range: \( 0 < p \leq 1 \)

**Hypergeometric Distribution**

\[
x = \text{RAND('HYPER',}N,R,n\text{'})
\]

where

\( x \)

is an integer observation from the distribution with the following probability density function:

\[
f(x) = \binom{R}{x} \binom{N-R}{n-x} / \binom{N}{n}
\]

Range: \( x = \max(0, (n - (N - R))), ..., \min(n, R) \)

\( N \)

is an integer population size parameter.

Range: \( N = 1, 2, ... \)

\( R \)

is an integer number of items in the category of interest.

Range: \( R = 0, 1, ..., N \)

\( n \)

is an integer sample size parameter.

Range: \( n = 1, 2, ..., N \)

The hypergeometric distribution is a mathematical formalization of an experiment in which you draw \( n \) balls from an urn that contains \( N \) balls, \( R \) of which are red. The hypergeometric distribution is the distribution of the number of red balls in the sample of \( n \).

**Lognormal Distribution**

\[
x = \text{RAND('LOGNORMAL')}
\]
where

\( x \)

is an observation from the distribution with the following probability density function:

\[
f (x) = \frac{e^{-\ln^2(x)/2}}{x \sqrt{2\pi}}
\]

**Range:** \( x \geq 0 \)

**Negative Binomial Distribution**

\( x = \text{RAND('NEGBINOMIAL',} p, k) \)

where

\( x \)

is an integer observation from the distribution with the following probability density function:

\[
f (x) = \begin{cases} \binom{x+k-1}{k-1} (1 - p)^x p^k & 0 < p < 1, x = 0, 1, \ldots \\ 1 & p = 1, x = 0 \\ \end{cases}
\]

**Range:** \( x = 0, 1, \ldots \)

\( k \)

is an integer number parameter that counts the number of successes.

**Range:** \( k = 1, 2, \ldots \)

\( p \)

is a numeric probability of success.

**Range:** \( 0 < p \leq 1 \)

The negative binomial distribution is the distribution of the number of failures before \( k \) successes occur in sequential independent trials, all with the same probability of success, \( p \).

**Normal Distribution**

\( x = \text{RAND('NORMAL',} \mu, \sigma) \)

where

\( x \)

is an observation from the normal distribution with a mean of \( \theta \) and a standard deviation of \( \lambda \), that has the following probability density function:

\[
f (x) = \frac{1}{\lambda \sqrt{2\pi}} e^{\exp \left( -\frac{(x - \theta)^2}{2\lambda^2} \right)}
\]

**Range:** \( -\infty < x < \infty \)
is the mean parameter.
**Default:** 0

\( \lambda \)
is the standard deviation parameter.
**Default:** 1
**Range:** \( \lambda > 0 \)

**Poisson Distribution**

\[ x = \text{RAND}('POISSON', m) \]

where

\( x \)
is an integer observation from the distribution with the following probability density function:

\[ f(x) = \frac{m^x e^{-m}}{x!} \]

**Range:** \( x = 0, 1, ... \)

\( m \)
is a numeric mean parameter.
**Range:** \( m > 0 \)

**T Distribution**

\[ x = \text{RAND}('T', df) \]

where

\( x \)
is an observation from the distribution with the following probability density function:

\[ f(x) = \frac{\Gamma\left(\frac{df+1}{2}\right)}{\sqrt{df \pi} \Gamma\left(\frac{df}{2}\right)} \left(1 + \frac{x^2}{df}\right)^{-\frac{df+1}{2}} \]

**Range:** \(-\infty < x < \infty\)

\( df \)
is a numeric degrees of freedom parameter.
**Range:** \( df > 0 \)
**Tabled Distribution**

\[ x = \text{RAND}('TABLE', p_1, p_2, ...) \]

where

\[ x \]

is an integer observation from one of the following distributions:

If \( \sum_{i=1}^{n} p_i \leq 1 \), then \( x \) is an observation from this probability density function:

\[
 f(i) = p_i, \quad i = 1, 2, \ldots, n \\
\text{and} \\
 f(n + 1) = 1 - \sum_{i=1}^{n} p_i
\]

If for some index \( \sum_{i=1}^{n} p_i \leq 1 \), then \( x \) is an observation from this probability density function:

\[
 f(i) = p_i, \quad i = 1, 2, \ldots, j - 1 \\
\text{and} \\
 f(j) = 1 - \sum_{i=1}^{j-1} p_i
\]

\[ p_1, p_2, \ldots \]

are numeric probability values.

**Range:** \( 0 \leq p_1, p_2, \ldots \leq 1 \)

**Restriction:** The maximum number of probability parameters depends on your operating environment, but the maximum number of parameters is at least 32,767.

The tabled distribution takes on the values 1, 2, ..., \( n \) with specified probabilities.

**Note:** By using the FORMAT statement, you can map the set \{1, 2, ..., \( n \}\) to any set of \( n \) or fewer elements.

**Triangular Distribution**

\[ x = \text{RAND}('TRIANGLE', h) \]

where

\[ x \]

is an observation from the distribution with the following probability density function:

\[
 f(x) = \begin{cases} 
 \frac{2x}{h} & 0 \leq x \leq h \\
 \frac{2(1-x)}{1-h} & h < x \leq 1
\end{cases}
\]

where \( 0 \leq h \leq 1 \).
Functions and CALL Routines

**Range:** $0 \leq x \leq 1$

*Note:* The distribution can be easily shifted and scaled. Δ

$h$

is the horizontal location of the peak of the triangle.

**Range:** $0 \leq h \leq 1$

**Uniform Distribution**

$x = \text{RAND}('\text{UNIFORM}')$

where

$x$

is an observation from the distribution with the following probability density function:

$$f(x) = 1$$

**Range:** $0 < x < 1$

The uniform random number generator that the RAND function uses is the Mersenne-Twister (Matsumoto and Nishimura 1998). This generator has a period of $2^{19937} - 1$ and 623-dimensional equidistribution up to 32-bit accuracy. This algorithm underlies the generators for the other available distributions in the RAND function.

**Weibull Distribution**

$x = \text{RAND}('\text{WEIBULL}',a,b)$

where

$x$

is an observation from the distribution with the following probability density function:

$$f(x) = \frac{a}{b^a} x^{a-1} e^{-\left(\frac{x}{b}\right)^a}$$

**Range:** $x \geq 0$

$a$

is a numeric shape parameter.

**Range:** $a > 0$

$b$

is a numeric scale parameter.

**Range:** $b > 0$

**Examples**

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=rand(‘BERN’, .75);</td>
<td>0</td>
</tr>
<tr>
<td>x=rand(‘BETA’, 3, 0.1);</td>
<td>.99920</td>
</tr>
<tr>
<td>x=rand(‘_BINOM’, 10, 0.75);</td>
<td>10</td>
</tr>
</tbody>
</table>
SAS Statements | Results
--- | ---
x=rand('CAUCHY'); | -1.41525
x=rand('CHISQ', 22); | 25.8526
x=rand('ERLANG', 7); | 7.67039
x=rand('EXPO'); | 1.48847
x=rand('F', 12, 322); | 1.99647
x=rand('GAMMA', 7.25); | 6.59588
x=rand('GEOM', 0.02); | 1
x=rand('HYPER', 10, 3, 5); | 0.66522
x=rand('LOGN'); | 0.66522
x=rand('NEGB', 5, 0.8); | 33
x=rand('NORMAL'); | 1.03507
x=rand('POISSON', 6.1); | 6
x=rand('T', 4); | 2.44646
x=rand('TABLE', .2, .5); | 2
x=rand('TRIANGLE', 0.7); | 0.63811
x=rand('UNIFORM'); | 0.96234
x=rand('WEIB', 0.25, 2.1); | 6.55778

See Also

CALL Routine:
“CALL STREAMINIT Routine” on page 400

References


RANEXP Function

Returns a random variate from an exponential distribution

Category: Random Number

Tip: If you want to change the seed value during execution, you must use the CALL RANEXP routine instead of the RANEXP function.

Syntax

RANEXP(seed)

Arguments

seed

is an integer. If seed ≤ 0, the time of day is used to initialize the seed stream.

Range: seed < 2\(^31\)−1

See: “Seed Values” on page 257 for more information about seed values

Details

The RANEXP function returns a variate that is generated from an exponential distribution with parameter 1. An inverse transform method applied to a RANUNI uniform variate is used.

An exponential variate X with parameter LAMBDA can be generated:

\[ x = \frac{\text{ranexp}(seed)}{\lambda} \]

An extreme value variate X with location parameter ALPHA and scale parameter BETA can be generated:

\[ x = \alpha - \beta \log(\text{ranexp}(seed)) \]

A geometric variate X with parameter P can be generated as follows:

\[ x = \lfloor -\frac{\text{ranexp}(seed)}{\log(1-P)} \rfloor \]

Comparisons

The CALL RANEXP routine, an alternative to the RANEXP function, gives greater control of the seed and random number streams.
See Also

Call routine:
“CALL RANEXP Routine” on page 369

RANGAM Function

Returns a random variate from a gamma distribution

Category: Random Number

Tip: If you want to change the seed value during execution, you must use the CALL RANGAM routine instead of the RANGAM function.

Syntax

RANGAM(seed,a)

Arguments

seed
is an integer. If seed ≤ 0, the time of day is used to initialize the seed stream.

Range: seed < \(2^{31}-1\)

See: “Seed Values” on page 257 for more information about seed values

\(\alpha\)

is a numeric shape parameter.

Range: \(\alpha > 0\)

Details

The RANGAM function returns a variate that is generated from a gamma distribution with parameter \(\alpha\). For \(\alpha > 1\), an acceptance-rejection method due to Cheng (1977) (See “References” on page 928) is used. For \(\alpha \leq 1\), an acceptance-rejection method due to Fishman is used (1978, Algorithm G2) (See “References” on page 928).

A gamma variate \(X\) with shape parameter \(\text{ALPHA}\) and scale \(\text{BETA}\) can be generated:

\[x = \text{beta} \times \text{rangam(seed, alpha)};\]

If \(2^n\text{\text{ALPHA}}\) is an integer, a chi-square variate \(X\) with \(2^n\text{\text{ALPHA}}\) degrees of freedom can be generated:

\[x = 2 \times \text{rangam(seed, alpha)};\]

If \(N\) is a positive integer, an Erlang variate \(X\) can be generated:

\[x = \text{beta} \times \text{rangam(seed, N)};\]

It has the distribution of the sum of \(N\) independent exponential variates whose means are \(\text{BETA}\).

And finally, a beta variate \(X\) with parameters \(\text{ALPHA}\) and \(\text{BETA}\) can be generated:
y1=rangam(seed, alpha);
y2=rangam(seed, beta);
x=y1/(y1+y2);

Comparisons
The CALL RANGAM routine, an alternative to the RANGAM function, gives greater
control of the seed and random number streams.

See Also
Call routine:
“CALL RANGAM Routine” on page 371

RANGE Function

Returns the range of values
Category: Descriptive Statistics

Syntax
RANGE(argument, argument, ...)

Arguments

argument
is numeric. At least one nonmissing argument is required. Otherwise, the function
returns a missing value. The argument list can consist of a variable list, which is
preceded by OF.

Details
The RANGE function returns the difference between the largest and the smallest of the
nonmissing arguments.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0=range(., .);</td>
<td>.</td>
</tr>
<tr>
<td>x1=range(-2, 6, 3);</td>
<td>8</td>
</tr>
<tr>
<td>x2=range(2, 6, 3, .);</td>
<td>4</td>
</tr>
<tr>
<td>x3=range(1, 6, 3, 1);</td>
<td>5</td>
</tr>
<tr>
<td>x4=range(of x1-x3);</td>
<td>4</td>
</tr>
</tbody>
</table>
**RANK Function**

Returns the position of a character in the ASCII or EBCDIC collating sequence

Category: Character

See: RANK Function in the documentation for your operating environment.

---

**Syntax**

\[ \text{RANK}(x) \]

**Arguments**

\( x \)

specifies a character expression.

**Details**

The RANK function returns an integer that represents the position of the first character in the character expression. The result depends on your operating environment.

**Examples**

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII</td>
<td>EBCDIC</td>
</tr>
<tr>
<td>n=rank(‘A’);</td>
<td>65</td>
</tr>
<tr>
<td>put n;</td>
<td></td>
</tr>
</tbody>
</table>

---

**See Also**

Functions:

“BYTE Function” on page 334

“COLLATE Function” on page 440

---

**RANNOR Function**

Returns a random variate from a normal distribution

Category: Random Number

Tip: If you want to change the seed value during execution, you must use the CALL RANNOR routine instead of the RANNOR function.
Syntax
RANNOR(seed)

Arguments

seed is an integer. If seed ≤ 0, the time of day is used to initialize the seed stream.
Range: seed < 2^{31} – 1
See: “Seed Values” on page 257 for more information about seed values

Details
The RANNOR function returns a variate that is generated from a normal distribution with mean 0 and variance 1. The Box-Muller transformation of RANUNI uniform variates is used.
A normal variate X with mean MU and variance S2 can be generated with this code:
\[ x = MU + \sqrt{S2} \times \text{rannor(seed)}; \]
A lognormal variate X with mean \( \exp(MU + S2/2) \) and variance \( \exp(2\times MU + 2\times S2) - \exp(2\times MU + S2) \) can be generated with this code:
\[ x = \exp(MU + \sqrt{S2} \times \text{rannor(seed)}); \]

Comparisons
The CALL RANNOR routine, an alternative to the RANNOR function, gives greater control of the seed and random number streams.

See Also
Call routine:
“CALL RANNOR Routine” on page 373

RANPOI Function

Returns a random variate from a Poisson distribution
Category: Random Number
Tip: If you want to change the seed value during execution, you must use the CALL RANPOI routine instead of the RANPOI function.

Syntax
RANPOI(seed, m)
Arguments

seed
is an integer. If seed ≤ 0, the time of day is used to initialize the seed stream.
Range: seed < 2^{31} − 1
See: “Seed Values” on page 257 for more information about seed values

m
is a numeric mean parameter.
Range: m ≥ 0

Details
The RANPOI function returns a variate that is generated from a Poisson distribution with mean m. For m < 85, an inverse transform method applied to a RANUNI uniform variate is used (Fishman 1976) (See “References” on page 928). For m ≥ 85, the normal approximation of a Poisson random variable is used. To expedite execution, internal variables are calculated only on initial calls (that is, with each new m).

Comparisons
The CALL RANPOI routine, an alternative to the RANPOI function, gives greater control of the seed and random number streams.

See Also
Call routine:
“CALL RANPOI Routine” on page 377

RANTBL Function
Returns a random variate from a tabled probability distribution
Category: Random Number
Tip: If you want to change the seed value during execution, you must use the CALL RANTBL routine instead of the RANTBL function.

Syntax
RANTBL(seed, p1, ..., pn)

Arguments

seed
is an integer. If seed ≤ 0, the time of day is used to initialize the seed stream.
Range: seed < 2^{31} − 1
See: “Seed Values” on page 257 for more information about seed values.

\( p_i \)
is a numeric SAS value.

**Range:** \( 0 \leq p_i \leq 1 \) for \( 0 < i \leq n \)

**Details**

The RANTBL function returns a variate that is generated from the probability mass function defined by \( p_i \) through \( p_n \). An inverse transform method applied to a RANUNI uniform variate is used. RANTBL returns

\[
\begin{align*}
1 & \text{ with probability } p_1 \\
2 & \text{ with probability } p_2 \\
& \ldots \\
& \ldots \\
n & \text{ with probability } p_n \\
n + 1 & \text{ with probability } 1 - \sum_{i=1}^{n} p_i \quad \text{if } \sum_{i=1}^{n} p_i \leq 1
\end{align*}
\]

If, for some index \( j < n \), \( \sum_{i=1}^{j} p_i \geq 1 \), RANTBL returns only the indices 1 through \( j \) with the probability of occurrence of the index \( j \) equal to \( 1 - \sum_{i=1}^{j-1} p_i \).

Let \( n=3 \) and \( P_1, P_2, \) and \( P_3 \) be three probabilities with \( P_1+P_2+P_3=1 \), and \( M_1, M_2, \) and \( M_3 \) be three variables. The variable \( X \) in these statements

```sas
array m{3} m1-m3;
x=m{rantbl(seed,of p1-p3)};
```

will be assigned one of the values of \( M_1, M_2, \) or \( M_3 \) with probabilities of occurrence \( P_1, P_2, \) and \( P_3 \), respectively.

**Comparisons**

The CALL RANTBL routine, an alternative to the RANTBL function, gives greater control of the seed and random number streams.

**See Also**

Call routine:

“CALL RANTBL Routine” on page 379

---

**RANTRI Function**

Returns a random variate from a triangular distribution.
Category: Random Number
Tip: If you want to change the seed value during execution, you must use the CALL RANTRI routine instead of the RANTRI function.

Syntax

RANTRI(seed, h)

Arguments

seed

is an integer. If seed ≤ 0, the time of day is used to initialize the seed stream.
Range: seed < 2^{31}−1
See: “Seed Values” on page 257 for more information about seed values

h

is a numeric SAS value.
range: 0 < h < 1

Details

The RANTRI function returns a variate that is generated from the triangular distribution on the interval (0,1) with parameter h, which is the modal value of the distribution. An inverse transform method applied to a RANUNI uniform variate is used.

A triangular distribution X on the interval [A,B] with mode C, where A ≤ C ≤ B, can be generated:

x=(b-a)*rantri(seed,(c-a)/(b-a))+a;

Comparisons

The CALL RANTRI routine, an alternative to the RANTRI function, gives greater control of the seed and random number streams.

See Also

Call routine:

“CALL RANTRI Routine” on page 381

RANUNI Function

Returns a random variate from a uniform distribution

Category: Random Number
Tip: If you want to change the seed value during execution, you must use the CALL RANUNI routine instead of the RANUNI function.
Syntax

RANUNI(seed)

Arguments

seed

is an integer. If seed ≤ 0, the time of day is used to initialize the seed stream.

Range: seed < 2^{31}−1

See: “Seed Values” on page 257 for more information about seed values

Details

The RANUNI function returns a number that is generated from the uniform distribution on the interval (0,1) using a prime modulus multiplicative generator with modulus 2^{31}−1 and multiplier 397204094 (Fishman and Moore 1982) (See “References” on page 928).

You can use a multiplier to change the length of the interval and an added constant to move the interval. For example,

```
random_variate=a*ranuni(seed)+b;
```

returns a number that is generated from the uniform distribution on the interval (b,a+b).

Comparisons

The CALL RANUNI routine, an alternative to the RANUNI function, gives greater control of the seed and random number streams.

See Also

Call routine:

“CALL RANUNI Routine” on page 383

REPEAT Function

Repeats a character expression

Category: Character

Syntax

REPEAT(argument,n)
Arguments

**argument**
specifies any SAS character expression.

**n**
specifies the number of times to repeat **argument**.

**Restriction:**  
**n** must be greater than or equal to 0.

Details

If the REPEAT function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

The REPEAT function returns a character value consisting of the first argument repeated **n** times. Thus, the first argument appears **n**+1 times in the result.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=repeat('ONE',2); put x;</td>
<td>ONEONEONE</td>
</tr>
</tbody>
</table>

**RESOLVE Function**

Returns the resolved value of an argument after it has been processed by the macro facility

**Category:** Macro

Syntax

RESOLVE(argument)

Arguments

**argument**
represents a macro expression.

Details

If the RESOLVE function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

RESOLVE is fully documented in *SAS Macro Language: Reference*.
See Also

Function:
“SYMGET Function” on page 845

REVERSE Function

Reverses a character expression

Category: Character

Syntax

REVERSE(argument)

Arguments

argument
specifies any SAS character expression.

Details

If the REVERSE function returns a value to a variable that has not yet been assigned a
length, by default the variable length is determined by the length of the first argument.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>backward=reverse('xyz ');</td>
<td>-------</td>
</tr>
<tr>
<td>put backward $5.;</td>
<td>zyx</td>
</tr>
</tbody>
</table>

REWIND Function

Positions the data set pointer at the beginning of a SAS data set

Category: SAS File I/O
Syntax

REWIND(data-set-id)

Arguments

data-set-id

specifies the data set identifier that the OPEN function returns.

Restriction: The data set cannot be opened in IS mode.

Details

REWIND returns 0 if the operation was successful, ≠0 if it was not successful. After a call to REWIND, a call to FETCH reads the first observation in the data set.

If there is an active WHERE clause, REWIND moves the data set pointer to the first observation that satisfies the WHERE condition.

Examples

This example calls FETCHOBS to fetch the tenth observation in the data set MYDATA, then calls REWIND to return to the first observation and fetch the first observation.

%let dsid=%sysfunc(open(mydata,i));
%let rc=%sysfunc(fetchobs(&dsid,10));
%let rc=%sysfunc(rewind(&dsid));
%let rc=%sysfunc(fetch(&dsid));

See Also

Functions:

“FETCH Function” on page 522
“FETCHOBS Function” on page 523
“FREWIND Function” on page 559
“NOTE Function” on page 671
“OPEN Function” on page 690
“POINT Function” on page 715
Arguments

*argument*

specifies any SAS character expression.

Details

If the RIGHT function returns a value to a variable that has not yet been assigned a length, by default the variable length is determined by the length of the first argument. The RIGHT function returns an argument with trailing blanks moved to the start of the value. The argument’s length does not change.

Examples

```
SAS Statements  Results
------+----1----+
  a='Due Date ';  
  b=right(a);     
  put a $10.;     
  put b $10.;     
      Due Date  
      Due Date
```

See Also

Functions:

“COMPRESS Function” on page 456
“LEFT Function” on page 625
“TRIM Function” on page 863

RMS Function

Returns the root mean square

Category: Descriptive Statistics

Syntax

RMS(argument<,argument,...>)
Arguments

**argument**
is a non-negative numeric constant, variable, or expression.

**Tip:** The argument list can consist of a variable list, which is preceded by `OF`.

Details

The root mean square is the square root of the arithmetic mean of the squares of the values. If all the arguments are missing values, then the result is a missing value. Otherwise, the result is the root mean square of the non-missing values.

Let \( n \) be the number of arguments with non-missing values, and let \( x_1, x_2, \ldots, x_n \) be the values of those arguments. The root mean square is

\[
\sqrt{\frac{x_1^2 + x_2^2 + \ldots + x_n^2}{n}}
\]

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x1=rms(1,7);</code></td>
<td>5</td>
</tr>
<tr>
<td><code>x2=rms(.,1,5,11);</code></td>
<td>7</td>
</tr>
<tr>
<td><code>x3=rms(of x1-x2);</code></td>
<td>6.0827625303</td>
</tr>
</tbody>
</table>

ROUND Function

Rounds the first argument to the nearest multiple of the second argument, or to the nearest integer when the second argument is omitted.

**Category:** Truncation

Syntax

`ROUND (argument <,rounding-unit>)`

Arguments

**argument**
is a numeric constant, variable, or expression to be rounded.

**rounding-unit**
is a positive, numeric constant, variable, or expression that specifies the rounding unit.

Details

Basic Concepts

The ROUND function rounds the first argument to a value that is very close to a multiple of the second argument. The results might not be an exact multiple of the second argument.

Differences between Binary and Decimal Arithmetic

Computers use binary arithmetic with finite precision. If you work with numbers that do not have an exact binary representation, computers often produce results that differ slightly from the results that are produced with decimal arithmetic.

For example, the decimal values 0.1 and 0.3 do not have exact binary representations. In decimal arithmetic, 3*0.1 is exactly equal to 0.3, but this equality is not true in binary arithmetic. As the following example shows, if you write these two values in SAS, they appear the same. If you compute the difference, however, you can see that the values are different.

```
data _null_; 
  point_three=0.3; 
  three_times_point_one=3*0.1; 
  difference=point_three - three_times_point_one; 
  put point_three= ; 
  put three_times_point_one= ; 
  put difference= ; 
run;
```

The following lines are written to the SAS log:

```
point_three= 0.3 
three_times_point_one= 0.3 
difference= 1.3878E-17 
```

Operating Environment Information: The example above was executed in a z/OS environment. If you use other operating environments, the results will be slightly different. △

The Effects of Rounding

Rounding by definition finds an exact multiple of the rounding unit that is closest to the value to be rounded. For example, 0.33 rounded to the nearest tenth equals 3*0.1 or 0.3 in decimal arithmetic. In binary arithmetic, 0.33 rounded to the nearest tenth equals 3*0.1, and not 0.3, because 0.3 is not an exact multiple of one tenth in binary arithmetic.

The ROUND function returns the value that is based on decimal arithmetic, even though this value is sometimes not the exact, mathematically correct result. In the example `ROUND(0.33,0.1)`, ROUND returns 0.3 and not 3*0.1.

Expressing Binary Values

If the characters "0.3" appear as a constant in a SAS program, the value is computed by the standard informat as 3/10. To be consistent with the standard informat, `ROUND(0.33,0.1)` computes the result as 3/10, and the following statement produces the results that you would expect.
if round(x,0.1) = 0.3 then
   ... more SAS statements ...

However, if you use the variable Y instead of the constant 0.3, as the following
statement shows, the results might be unexpected depending on how the variable Y is
computed.

if round(x,0.1) = y then
   ... more SAS statements ...

If ROUND reads Y as the characters "0.3" using the standard informat, the result is
the same as if a constant 0.3 appeared in the IF statement. If ROUND reads Y with a
different informat, or if a program other than SAS reads Y, then there is no guarantee
that the characters "0.3" would produce a value of exactly 3/10. Imprecision can also be
cased by computation involving numbers that do not have exact binary
representations, or by porting data sets from one operating environment to another that
has a different floating-point representation.

If you know that Y is a decimal number with one decimal place, but are not certain
that Y has exactly the same value as would be produced by the standard informat, it is
better to use the following statement:

if round(x,0.1) = round(y,0.1) then
   ... more SAS statements ...

Testing for Approximate Equality

You should not use the ROUND function as a general method to test for approximate
equality. Two numbers that differ only in the least significant bit can round to different
values if one number rounds down and the other number rounds up. Testing for
approximate equality depends on how the numbers have been computed. If both
numbers are computed to high relative precision, you could test for approximate
equality by using the ABS and the MAX functions, as the following example shows.

if abs(x-y) <= 1e-12 * max( abs(x), abs(y) ) then
   ... more SAS statements ...

Producing Expected Results

In general, ROUND(argument, rounding-unit) produces the result that you expect
from decimal arithmetic if the result has no more than nine significant digits and any of
the following conditions are true:

- The rounding unit is an integer.
- The rounding unit is a power of 10 greater than or equal to 1e-15. *
- The result that you expect from decimal arithmetic has no more than four decimal
  places.

For example:

data rounding;
   d1 = round(1234.56789,100) - 1200;
   d2 = round(1234.56789,10)  - 1230;
   d3 = round(1234.56789,1)   - 1235;
   d4 = round(1234.56789,.1)  - 1234.6;

* If the rounding unit is less than one, ROUND treats it as a power of 10 if the reciprocal of the rounding unit differs from a
  power of 10 in at most the three or four least significant bits.
Functions and CALL Routines

ROUND Function

\[
d_5 = \text{round}(1234.56789,.01) - 1234.57;
\]
\[
d_6 = \text{round}(1234.56789,.001) - 1234.568;
\]
\[
d_7 = \text{round}(1234.56789,.0001) - 1234.5679;
\]
\[
d_8 = \text{round}(1234.56789,.00001) - 1234.56789;
\]
\[
d_9 = \text{round}(1234.56789,.1111) - 1234.5432;
\]
\[
/* d10 has too many decimal places in the value for */
/* rounding-unit. */
\]
\[
d_{10} = \text{round}(1234.56789,.11111) - 1234.54321;
\]
run;

proc print data=rounding noobs;
run;

The following output shows the results.

Output 4.41  Results of Rounding Based on the Value of the Rounding Unit

<table>
<thead>
<tr>
<th></th>
<th>The SAS System</th>
</tr>
</thead>
<tbody>
<tr>
<td>d1</td>
<td>0</td>
</tr>
<tr>
<td>d2</td>
<td>0</td>
</tr>
<tr>
<td>d3</td>
<td>0</td>
</tr>
<tr>
<td>d4</td>
<td>0</td>
</tr>
<tr>
<td>d5</td>
<td>0</td>
</tr>
<tr>
<td>d6</td>
<td>0</td>
</tr>
<tr>
<td>d7</td>
<td>0</td>
</tr>
<tr>
<td>d8</td>
<td>0</td>
</tr>
<tr>
<td>d9</td>
<td>0</td>
</tr>
<tr>
<td>d10</td>
<td>-2.2737E-13</td>
</tr>
</tbody>
</table>

Operating Environment Information:  The example above was executed in a z/OS environment. If you use other operating environments, the results will be slightly different.

When the Rounding Unit Is the Reciprocal of an Integer

When the rounding unit is the reciprocal of an integer **, the ROUND function computes the result by dividing by the integer. Therefore, you can safely compare the result from ROUND with the ratio of two integers, but not with a multiple of the rounding unit. For example:

\[
data \text{ rounding2;}
\]
\[
drop \ pi \ unit;
\]
\[
pi = \text{arccos}(-1);
\]
\[
unit=1/7;
\]
\[
d1=\text{round}(pi,unit) - 22/7;
\]
\[
d2=\text{round}(pi, unit) - 22*unit;
\]
run;

proc print data=rounding2 noobs;
run;

The following output shows the results.

\** ROUND treats the rounding unit as a reciprocal of an integer if the reciprocal of the rounding unit differs from an integer in at most the three or four least significant bits.**
Output 4.42  Results of Rounding by the Reciprocal of an Integer

<table>
<thead>
<tr>
<th>The SAS System</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>d1</td>
<td>d2</td>
</tr>
<tr>
<td>0</td>
<td>2.2204E-16</td>
</tr>
</tbody>
</table>

Operating Environment Information: The example above was executed in an z/OS environment. If you use other operating environments, the results will be slightly different. △

Computing Results in Special Cases

The ROUND function computes the result by multiplying an integer by the rounding unit when all of the following conditions are true:

- The rounding unit is not an integer.
- The rounding unit is not a power of 10.
- The rounding unit is not the reciprocal of an integer.
- The result that you expect from decimal arithmetic has no more than four decimal places.

For example:

```sas
options pageno=1 nodate ls=80 ps=64;
data _null_;  
difference=round(1234.56789,.11111) - 11111*.11111;  
put difference=;  
run;
```

The following line is written to the SAS log:

```sas
difference=0
```

Operating Environment Information: The example above was executed in a z/OS environment. If you use other operating environments, the results might be slightly different. △

Computing Results When the Value Is Halfway between Multiples of the Rounding Unit

When the value to be rounded is approximately halfway between two multiples of the rounding unit, the ROUND function rounds up the absolute value and restores the original sign. For example:

```sas
options pageno=1 nodate ls=80 ps=64;
data test;  
do i=8 to 17;  
  value=0.5 - 10**(-i);  
  round=round(value);  
  output;  
end;  
do i=8 to 17;  
  value=-0.5 + 10**(-i);  
  round=round(value);  
  output;
```
The following output shows the results.

Output 4.43  Results of Rounding When Values Are Halfway between Multiples of the Rounding Unit

<table>
<thead>
<tr>
<th></th>
<th>The SAS System</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>i</td>
<td>value</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.49999999900000000000000</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>0.49999999990000000000000</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0.49999999999000000000000</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>0.49999999999900000000000</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>0.49999999999990000000000</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>0.49999999999999000000000</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>0.49999999999999900000000</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>0.49999999999999990000000</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>0.50000000000000000000000</td>
</tr>
<tr>
<td></td>
<td>17</td>
<td>0.50000000000000000000000</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>-0.49999999900000000000000</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>-0.49999999990000000000000</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>-0.49999999999000000000000</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>-0.49999999999900000000000</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>-0.49999999999990000000000</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>-0.49999999999999000000000</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>-0.49999999999999900000000</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>-0.49999999999999990000000</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>-0.50000000000000000000000</td>
</tr>
<tr>
<td></td>
<td>17</td>
<td>-0.50000000000000000000000</td>
</tr>
</tbody>
</table>

Operating Environment Information:  The example above was executed in a z/OS environment. If you use other operating environments, the results might be slightly different.

The approximation is relative to the size of the value to be rounded, and is computed in a manner that is shown in the following DATA step. This DATA step code will not always produce results exactly equivalent to the ROUND function.

```
data testfile;
  do i = 1 to 17;
    value = 0.5 -- 10**(-i);
    epsilon = min(1e-6, value * 1e-12);
    temp = value + .5 + epsilon;
    fraction = modz(temp, 1);
    round = temp - fraction;
    output;
  end;
run;
```

Comparisons

The ROUND function is the same as the ROUNDE function except that when the first argument is halfway between the two nearest multiples of the second argument,
ROUND returns an even multiple. ROUND returns the multiple with the larger absolute value.

The ROUNDZ function returns a multiple of the rounding unit without trying to make the result match the result that is computed with decimal arithmetic.

**Examples**

The following example compares the results that are returned by the ROUND function with the results that are returned by the ROUNDZ function. The output was generated from the UNIX operating environment.

```sas
options pageno=1 nodate linesize=80 pagesize=60;

data results;
  do x=0 to 4 by .25;
    Rounde=rounde(x);
    Round=round(x);
    output;
  end;
run;

proc print data=results noobs;
run;
```

The following output shows the results.

**Output 4.44 Results That Are Returned by the ROUND and ROUNDZ Functions**

<table>
<thead>
<tr>
<th>x</th>
<th>Rounde</th>
<th>Round</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0.25</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0.50</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0.75</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1.00</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1.25</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1.50</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1.75</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2.00</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2.25</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2.50</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2.75</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3.00</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3.25</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3.50</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3.75</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4.00</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

**See Also**

Functions:

“CEIL Function” on page 429
“CEILZ Function” on page 430
“FLOOR Function” on page 544
“FLOORZ Function” on page 545
ROUNDE Function

Rounds the first argument to the nearest multiple of the second argument, and returns an even multiple when the first argument is halfway between the two nearest multiples.

Category: Truncation

Syntax

ROUNDE (argument < , rounding-unit > )

Arguments

argument

is a numeric constant, variable, or expression to be rounded.

rounding-unit

is a positive, numeric constant, variable, or expression that specifies the rounding unit.

Details

The ROUNDE function rounds the first argument to the nearest multiple of the second argument. If you omit the second argument, ROUNDE uses a default value of 1 for rounding-unit.

Comparisons

The ROUNDE function is the same as the ROUND function except that when the first argument is halfway between the two nearest multiples of the second argument, ROUNDE returns an even multiple. ROUND returns the multiple with the larger absolute value.

Examples

The following example compares the results that are returned by the ROUNDE function with the results that are returned by the ROUND function.

```plaintext
options pageno=1 nodate linesize=80 pagesize=60;

data results;
  do x=0 to 4 by .25;
    Rounde=rounde(x);
    Round=round(x);
  
```
output;
end;
run;

proc print data=results noobs;
run;

The following output shows the results.

### Output 4.45  Results That are Returned by the ROUND and ROUND Functions

<table>
<thead>
<tr>
<th>x</th>
<th>Rounde</th>
<th>Round</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0.25</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0.50</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0.75</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1.00</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1.25</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1.50</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1.75</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2.00</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2.25</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2.50</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2.75</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3.00</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3.25</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3.50</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3.75</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4.00</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

### See Also

Function:

- “CEIL Function” on page 429
- “CEILZ Function” on page 430
- “FLOOR Function” on page 544
- “FLOORZ Function” on page 545
- “INT Function” on page 597
- “INTZ Function” on page 608
- “ROUND Function” on page 790
- “ROUNDZ Function” on page 798

### ROUNDZ Function

Rounds the first argument to the nearest multiple of the second argument, with zero fuzzing

**Category:** Truncation
Syntax
ROUNDZ (argument <,rounding-unit>)

Arguments

argument
is a numeric constant, variable, or expression to be rounded.

rounding-unit
is a positive, numeric constant, variable, or expression that specifies the rounding unit.

Details
The ROUNDZ function rounds the first argument to the nearest multiple of the second argument. If you omit the second argument, ROUNDZ uses a default value of 1 for rounding-unit.

Comparisons
The ROUNDZ function is the same as the ROUND function except that:

- ROUNDZ returns an even multiple when the first argument is exactly halfway between the two nearest multiples of the second argument. ROUND returns the multiple with the larger absolute value when the first argument is approximately halfway between the two nearest multiples.
- When the round-off unit is less than one and not the reciprocal of an integer, the result that is returned by ROUNDZ might not agree exactly with the result from decimal arithmetic. ROUNDZ does not fuzz the result. ROUND performs extra computations, called fuzzing, to try to make the result agree with decimal arithmetic.

Examples

Example 1: Comparing Results from the ROUNDZ and ROUND Functions
The following example compares the results that are returned by the ROUNDZ and the ROUND function.

```plaintext
options pageno=1 nodate linesize=60 pagesize=60;

data test;
od i=10 to 17;
   Value=2.5 - 10**(-i);
   Roundz=roundz(value);
   Round=round(value);
   output;
end;
do i=16 to 12 by -1;
   value=2.5 + 10**(-i);
   roundz=roundz(value);
   round=round(value);
   output;
end;
run;
```
proc print data=test noobs;
   format value 19.16;
run;

The following output shows the results.

Output 4.46  Results That Are Returned by the ROUNDZ and ROUND Functions

<table>
<thead>
<tr>
<th>i</th>
<th>Value</th>
<th>Roundz</th>
<th>Round</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>2.499999999999000000</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>2.499999999999900000</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>2.499999999999990000</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>2.499999999999999000</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>14</td>
<td>2.499999999999999999</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>15</td>
<td>2.499999999999999999</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td>2.500000000000000000</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>17</td>
<td>2.500000000000000000</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td>2.500000000000000000</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>15</td>
<td>2.500000000000000100</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>14</td>
<td>2.500000000000010000</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>2.500000000000100000</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>2.500000000001000000</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Example 2: Sample Output from the ROUNDZ Function

These examples show the results that are returned by ROUNDZ.

<table>
<thead>
<tr>
<th>SAS Statement</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>var1=223.456; x=roundz(var1,1);</td>
<td></td>
</tr>
<tr>
<td>put x 9.5;</td>
<td>223.00000</td>
</tr>
<tr>
<td>var2=223.456; x=roundz(var2,.01);</td>
<td></td>
</tr>
<tr>
<td>put x 9.5;</td>
<td>223.46000</td>
</tr>
<tr>
<td>x=roundz(223.456,100);</td>
<td></td>
</tr>
<tr>
<td>put x 9.5;</td>
<td>200.00000</td>
</tr>
<tr>
<td>x=roundz(223.456);</td>
<td></td>
</tr>
<tr>
<td>put x 9.5;</td>
<td>223.00000</td>
</tr>
<tr>
<td>x=roundz(223.456,.3);</td>
<td></td>
</tr>
<tr>
<td>put x 9.5;</td>
<td>223.50000</td>
</tr>
</tbody>
</table>

See Also

Functions:

“ROUND Function” on page 790
“ROUNDE Function” on page 797
RXMATCH Function

Finds the beginning of a substring that matches a pattern

Category: Character String Matching
Restriction: Use with the RXPARSE function

Syntax

\[ \text{position} = \text{RXMATCH}(\text{rx, string}) \]

Arguments

- **position**: specifies a numeric position in a string where the substring that is matched by the pattern begins. If there is no match, the result is zero.
- **rx**: specifies a numeric value that is returned from the RXPARSE function.
- **string**: specifies the character expression to be searched.

Details

RXMATCH searches the variable string for the pattern from RXPARSE and returns the position of the start of the string.

Comparisons

The regular expression (RX) functions and CALL routines work together to manipulate strings that match patterns. Use the RXPARSE function to parse a pattern you specify. Use the RXMATCH function and the CALL RXCHANGE and CALL RXSUBSTR routines to match or modify your data. Use the CALL RXFREE routine to free allocated space.

Example

See the RXPARSE function “Examples” on page 812.

See Also

- Functions and CALL routines:
  - “CALL RXCHANGE Routine” on page 385
  - “CALL RXFREE Routine” on page 386
  - “RXPARSE Function” on page 802
  - “CALL RXSUBSTR Routine” on page 387
RXPARSE Function

Parses a pattern

Category: Character String Matching

Syntax

```
rx = RXPARSE(pattern-expression)
```

Syntax Description

Arguments

`rx`

specifies a numeric value that is passed to other regular expression (RX) functions and CALL routines.

`pattern-expression`

specifies a character constant, variable, or expression whose value is a literal or a pattern expression. A `pattern-expression` is composed of the following elements:

- **string-in-quotation-marks**
  - matches a substring consisting of the characters in the string.

- **letter**
  - matches the uppercase or lowercase letter in a substring.

- **digit**
  - matches the digit in a substring.

- **period (.)**
  - matches a period (.) in a substring.

- **underscore (_)**
  - matches an underscore (_) in a substring.

- **?**
  - matches any one character in a substring.

- **colon (:)**
  - matches any sequence of zero or more characters in a substring.

- **$'pattern’ or "$pattern"**
  - matches any one character in a substring.

**Tip:** Ranges of alphanumeric variables are indicated by the hyphen (-).

**Example:** To match any lowercase letter, use

```
rx = rxparse("$'a-z';
```

**See:** “User-defined Character Classes” on page 805

- `~'character-class’ or ^'character-class’ or "character-class" or ^"character-class"
  - matches any one character that is not matched by the corresponding character class.

**Tip:** this is a summary definition. Check alignment.
Tip: Ranges of alphanumeric variables are indicated by a hyphen (-).

Example: To exclude the letters a-d from the match, use

```plaintext
rx=rxpathes("^'a-d'");
```

See: “Character Class Complements” on page 806

```plaintext
pattern1 pattern2 or pattern1 || pattern2
```

selects any substring matched by pattern1 followed immediately by any substring matched by pattern2 (with no intervening blanks).

```plaintext
pattern1 || pattern2
```

selects any substring matched by pattern1 or any substring matched by pattern2.

Tip: You can use an exclamation point (!) instead of a vertical bar (|).

```plaintext
(pattern)
```

matches a substring that contains a pattern. You can use parentheses to indicate the order in which operations are performed.

```plaintext
[pattern] or {pattern}
```

matches a substring that contains a pattern or null string.

```plaintext
pattern*
```

matches zero or more consecutive strings matched by a pattern.

```plaintext
pattern+
```

matches one or more consecutive strings matched by a pattern.

```plaintext
@int
```

matches the position of a variable if the next character is located in the column specified by `int`. @0 matches end-of-line. If `int` is negative, it matches -`int` positions from end-of-line.

```plaintext
reuse-character-class
```

reuses a `character-class` you previously defined.

See: “Reusing Character Classes” on page 807

```plaintext
pattern-abbreviation
```

specifies ways to shorten pattern representation.

See: “Pattern Abbreviations” on page 807, “Default Character Classes” on page 804

```plaintext
balanced-symbols
```

specifies the number of nested parentheses, brackets, braces, or less-than/greater-than symbols in a mathematical expression.

See: “Matching Balanced Symbols” on page 808

```plaintext
special-symbol
```

specifies a position in a string, or a score value.

See: “Special Symbols” on page 809

```plaintext
score-value
```

selects the pattern with the highest score value.

See: “Scores” on page 810

```plaintext
<pattern>
```

retrieves a matched substring for use in a change expression.

See: “Tag Expression” on page 810
change-expression
 specifies a pattern change operation that replaces a string containing a matched substring by concatenating values to the replacement string.

See: “Change Expressions” on page 811

change-item
 specifies items used for string manipulation.

See: “Change Items” on page 811

Character Classes

Using a character class element is a shorthand method for specifying a range of values for matching. In pattern matching, you can

- use default character classes
- define your own character classes
- use character class complements
- reuse character classes.

Default Character Classes

You specify a default character class with a dollar sign ($) followed by a single uppercase or lowercase letter. In the following list, the character class is listed in the left column and the definition is listed in the right column.

- $a$ or $A$ matches any alphabetic uppercase or lowercase letter in a substring ($'a-zA-Z'$).
- $c$ or $C$ matches any character allowed in a version 6 SAS name that is found in a substring ($'0-9a-zA-Z_.'$).
- $d$ or $D$ matches any digit in a substring ($'0-9'$).
- $i$ or $I$ matches any initial character in a version 6 SAS name that is found in a substring ($'a-zA-Z_.'$).
- $l$ or $L$ matches any lowercase letter in a substring ($'a-z'$).
- $u$ or $U$ matches any uppercase letter in a substring ($'A-Z'$).
- $w$ or $W$ matches any white space character, such as blank, tab, backspace, carriage return, etc., in a substring.

See also: “Character Class Complements” on page 806

Note: A hyphen appearing at the beginning or end of a character class is treated as a member of the class rather than as a range symbol. △

This statement and these values produce these matches.

```
rx=rxparse("$character-class$");
```

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Input string</th>
<th>Position of match</th>
<th>Value of match</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L$ or $l$</td>
<td>3+Y STRIkeS</td>
<td>9</td>
<td>k</td>
</tr>
<tr>
<td>$U$ or $u$</td>
<td>0*5x49XY</td>
<td>7</td>
<td>X (uppercase)</td>
</tr>
</tbody>
</table>
The following example shows how to use a default character class in a DATA step.

data _null_
stringA='3+Y STRIkeS';
rx=rxparse("$L");
matchA = rxmatch(rx,stringA);
valueA=substr(stringA,matchA,1);
put 'Example A: ' matchA = valueA= ;
run;

data _null_
stringA2='0*5x49XY';
rx=rxparse("$u");
matchA2 = rxmatch(rx,stringA2);
valueA2 = substr(stringA2,matchA2,1);
put 'Example A2: ' matchA2 = valueA2= ;
run;

The SAS log shows the following results:

Example A: matchA=9 valueA=k
Example A2: matchA2=7 valueA2=X

**User-defined Character Classes**   A user-defined character class begins with a dollar sign ($) and is followed by a string in quotation marks. A character class matches any one character within the quotation marks.

*Note:* Ranges of values are indicated by a hyphen (-).

This statement and these values produce these matches.

rx=rxparse("$'pattern'");

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Input string</th>
<th>Position of match</th>
<th>Value of match</th>
</tr>
</thead>
<tbody>
<tr>
<td>$'abcde'</td>
<td>3+yE strikes</td>
<td>11</td>
<td>e</td>
</tr>
<tr>
<td>$’1-9’</td>
<td>z0*549xy</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

The following example shows how to use a user-defined character class in a DATA step.

data _null_
stringB='3+yE strikes';
rx=rxparse("$’abcde’");
matchB = rxmatch(rx,stringB);
valueB=substr(stringB,matchB,1);
put 'Example B: ' matchB = valueB= ;
run;

data _null_
stringB2='z0*549xy';
rx=rxparse("$’1-9’");
matchB2=rxmatch(rx,stringB2);
valueB2=substr(stringB2,matchB2,1);
put 'Example B2: ' matchB2 = valueB2= ;
run;
The SAS log shows the following results:

Example B: matchB=11 valueB=e
Example B2: matchB2=4 valueB2=5

You can also define your own character class complements.

See: “Character Class Complements” on page 806 for details about character class complements

**Character Class Complements** A character class complement begins with a caret (^) or a tilde (~) and is followed by a string in quotation marks. A character class complement matches any one character that is *not* matched by the corresponding character class.

See: “Character Classes” on page 804 for details about character classes

This statement and these values produce these matches.

\[
\text{rx}=\text{rxparse}('\text{character-class} | -\text{character-class});
\]

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Input string</th>
<th>Position of match</th>
<th>Value of match</th>
</tr>
</thead>
<tbody>
<tr>
<td>^u or -u</td>
<td>0*5x49XY</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>^'A-z' or ~'A-z'</td>
<td>Abc de45</td>
<td>4</td>
<td>the first space</td>
</tr>
</tbody>
</table>

The following example shows how to use a character class complement in a DATA step.

```sas
data _null_;
  stringC='0*5x49XY';
  rx=rxparse('^u');
  matchC = rxmatch(rx,stringC);
  valueC=substr(stringC,matchC,1);
  put 'Example C: ' matchC = valueC=;
run;

data _null_;
  stringC2='Abc de45';
  rx=rxparse('~'-'A-z-');
  matchC2=rxmatch(rx,stringC2);
  valueC2=substr(stringC2,matchC2,1);
  put 'Example C2: ' matchC2 = valueC2= ;
run;
```

The SAS log shows the following results:

Example C: matchC=1 valueC=0
Example C2: matchC2=4 valueC2=
Reusing Character Classes  You can reuse character classes you previously defined by using one of the following patterns:

$\text{int}$  
reuses the $\text{int}^{th}$ character class.  

**Restriction:**  $\text{int}$ is a non-zero integer.  

**Example:**  If you defined a character class in a pattern and want to use the same character class again in the same pattern, use $\text{int}$ to refer to the $\text{int}^{th}$ character class you defined. If $\text{int}$ is negative, count backwards from the last pattern to identify the character class for $-\text{int}$. For example,

```plaintext
rx=rxpatharse("$'AB' \$1 $'XYZ' \$2 $-2");
```

is equivalent to

```plaintext
rx=rxpatharse("$'AB' $'AB' $'XYZ' $'XYZ' $'AB'");
```

- The $\$1$ element in the first code sample is replaced by AB in the second code sample, because AB was the first pattern defined.  
- The $\$2$ element in the first code sample is replaced by XYZ in the second code sample, because XYZ was the second pattern defined.  
- The $\$-2$ element in the first code sample is replaced by AB in the second code sample, because AB is the second-to-the-last pattern defined.  

$\neg\text{int}$ or $^\text{int}$  
reuses the complement of the $\text{int}^{th}$ character class.  

**Restriction:**  $\text{int}$ is a non-zero integer.  

**Example:**  This example shows character-class elements ($\$'Al'$, $\$'Jo'$, $\$'Li'$) and reuse numbers ($1$, $2$, $3$, $-2$):

```plaintext
rx=rxpatharse($'Al' \$1 $'Jo' \$2 $'Li' \$3 -2);
```

is equivalent to

```plaintext
rx=rxpatharse($'Al' $'Al' $'Jo' $'Jo'$ $'Li' $'Li' $'Al' $'Li'$);
```

The $\neg2$ matches patterns 1 (Al) and 3 (Li), and excludes pattern 2 (Jo).

**Pattern Abbreviations**

You can use the following list of elements in your pattern:

$\$f$ or $\$F$  matches a floating-point number.  
$\$n$ or $\$N$  matches a SAS name.  
$\$p$ or $\$P$  indicates a prefix option.  
$\$q$ or $\$Q$  matches a string in quotation marks.  
$\$s$ or $\$S$  indicates a suffix option.  

This statement and input string produce these matches.

```plaintext
rx=rxpatharse($\text{pattern-abbreviation pattern}$);
```
The following example shows how to use a pattern abbreviation in a DATA step.

data _null_
stringD='woodchucks eat firewood';
rx=rxparse("$p 'wood' ");
PositionOfMatchD=rxmatch(rx,stringD);
call rxsubstr(rx,stringD,PositionOfMatchD,lengthD);
valueD=substr(stringD,PositionOfMatchD);
put 'Example D: ' lengthD= valueD= ;
run;

data _null_
stringD2='woodchucks eat firewood';
rx=rxparse("'wood' $s ");
PositionOfMatchD2=rxmatch(rx,stringD2);
call rxsubstr(rx,stringD2,PositionOfMatchD2,lengthD2);
valueD2=substr(stringD2,PositionOfMatchD2);
put 'Example D2: ' lengthD2= valueD2= ;
run;

The SAS log shows the following results:

Example D: lengthD=4 valueD=woodchucks eat firewood
Example D2: lengthD2=4 valueD2=wood

**Matching Balanced Symbols**

You can match mathematical expressions containing multiple sets of balanced parentheses, brackets, braces, and less-than/greater-than symbols. Both the symbols and the expressions within the symbols are part of the match:

\[
$(int) \text{ or } $(int) \text{ or } $(int) \text{ or } $(int) >
\]

indicates the int level of nesting you specify.

**Restriction:** int is a positive integer.

**Tip:** Using smaller values increases the efficiency of finding a match.

**Example:** This statement and input string produces this match.

\[
rx=rxparse("$(2) ");
\]

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Input string</th>
<th>Position of match</th>
<th>Value of match</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p \text{ wood}</td>
<td>woodchucks eat wood</td>
<td>1</td>
<td>characters &quot;wood&quot; in woodchucks</td>
</tr>
<tr>
<td>\textbf{wood} $s</td>
<td>woodchucks eat wood</td>
<td>20</td>
<td>\textbf{wood}</td>
</tr>
<tr>
<td>(((a+b)*5)/43)</td>
<td>2</td>
<td>(((a+b)*5))</td>
<td></td>
</tr>
</tbody>
</table>
The following example shows how to use mathematical symbol matching in a DATA step.

```sas
data _null_
  stringE='(((a+b)*5)/43)';
  rx=rxparse("$(2)" culturally);
  call rxsubstr(rx,stringE,positionE,lengthE);
  PositionOfMatchE=rxmatch(rx,stringE);
  valueE=substr(stringE,PositionOfMatchE);
  put 'Example E: ' lengthE= valueE= ;
run;
```

The SAS log shows the following results:

Example E: lengthE=9 valueE= ((a+b)*5)/43)

### Special Symbols

You can use the following list of special symbols in your pattern:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\</td>
<td>sets the beginning of a match to the current position.</td>
</tr>
<tr>
<td>/</td>
<td>sets the end of a match to the current position.</td>
</tr>
<tr>
<td>$#</td>
<td>requests the match with the highest score, regardless of the starting position. <strong>Tip:</strong> The position of this symbol within the pattern is not significant.</td>
</tr>
<tr>
<td>$-</td>
<td>scans a string from right to left. <strong>Tip:</strong> The position of this symbol within the pattern is not significant.</td>
</tr>
<tr>
<td>$@</td>
<td>requires the match to begin where the scan of the text begins. <strong>Tip:</strong> The position of this symbol within the pattern is not significant.</td>
</tr>
</tbody>
</table>

The following table shows how a pattern matches an input string.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Input string</th>
<th>Value of match</th>
</tr>
</thead>
<tbody>
<tr>
<td>c\ow</td>
<td>How now brown cow?</td>
<td>characters “ow” in cow</td>
</tr>
<tr>
<td>ow/n</td>
<td>How now brown cow?</td>
<td>characters “ow” in brown</td>
</tr>
<tr>
<td>@3:\ow</td>
<td>How now brown cow?</td>
<td>characters “ow” in now</td>
</tr>
</tbody>
</table>
The following example shows how to use special symbol matching in a DATA step.

```sas
data _null_;  
  stringF='How now brown cow?';  
  rx=rxparse("$'c\ow'’");  
  matchF=rxmatch(rx,stringF);  
  valueF=substr(stringF,matchF,2);  
  put ‘Example F= ’ matchF= valueF= ;  
run;

data _null_;  
  stringF2='How now brown cow?';  
  rx=rxparse("@3:\ow”");  
  matchF2=rxmatch(rx,stringF2);  
  valueF2=substr(stringF2,matchF2,2);  
  put ‘Example F2= ’ matchF2= valueF2= ;  
run;
```

The SAS log shows the following results:

```
Example F= matchF=2 valueF=ow  
Example F2= matchF2=6 valueF2=ow
```

**Scores**

When a pattern is matched by more than one substring beginning at a specific position, the longest substring is selected. To change the selection criterion, assign a score value to each substring by using the pound sign (#) special symbol followed by an integer.

The score for any substring begins at zero. When #int is encountered in the pattern, the value of int is added to the score. If two or more matching substrings begin at the same leftmost position, SAS selects the substring with the highest score value. If two substrings begin at the same leftmost position and have the same score value, SAS selects the longer substring. The following is a list of score representations:

- `#int` adds int to the score, where int is a positive or negative integer.
- `#*int` multiplies the score by nonnegative int.
- `#/int` divides the score by positive int.
- `#=int` assigns the value of int to the score.
- `#>int` finds a match if the current score exceeds int.

**Tag Expression**

You can assign a substring of the string being searched to a character variable with the expression `name=`pattern`, where pattern specifies any pattern expression. The substring matched by this expression is assigned to the variable name.

If you enclose a pattern in less-than/greater-than symbols (<>) and do not specify a variable name, SAS automatically assigns the pattern to a variable. SAS assigns the variable _1 to the first occurrence of the pattern, _2 to the second occurrence, etc. This assignment is called tagging. SAS tags the corresponding substring of the matched string.

The following shows the syntax of a tag expression:

```
<pattern>
```

specifies a pattern expression. SAS assigns a variable to each occurrence of pattern for use in a change expression.
**Change Expressions**

If you find a substring that matches a pattern, you can change the substring to another value. You must specify the pattern expression, use the TO keyword, and specify the change expression in the argument for RXPARSE. You can specify a list of pattern change expressions by separating each expression with a comma.

A pattern change operation replaces a matched string by concatenating values to the replacement string. The operation concatenates

- all characters to the left of the match
- the characters specified in the change expression
- all characters to the right of the match.

You can have multiple parallel operations within the RXPARSE argument. In the following example,

```c
rx=rxparse("x TO y, y TO x");
```
x in a substring is substituted for y, and y in a substring is substituted for x.

A change expression can include the items in the following list. Each item in the list is followed by the description of the value concatenated to the replacement string at the position of the pointer:

- **string in quotation marks**
  - concatenates the contents of the string.

- **name**
  - concatenates the name, possibly in a different case.

- **number**
  - concatenates the number.

- **period (.)**
  - concatenates the period (.).

- **underscore ( underscore)**
  - concatenates the underscore (_).

- **=int**
  - concatenates the value of the int<sup>th</sup> tagged substring if int is positive, or the -int<sup>th</sup>-from-the-last tagged substring if int is negative. In a parallel change expression, the int<sup>th</sup> or -int<sup>th</sup>-from-the-last tag is counted within the component of the parallel change expression that yielded the match, and not over the entire parallel change expression.

- **==**
  - concatenates the entire matched substring.

**Change Items**

You can use the items in the following list to manipulate the replacement string. The items position the cursor without affecting the replacement string.

- **@int**
  - moves the pointer to column int where the next string added to the replacement string will start.

- **=**
  - moves the pointer one column past the end of the matched substring.

- **>int**
  - moves the pointer to the right to column int. If the pointer is already to the right of column int, the pointer is not moved.

- **>=**
  - moves the pointer to the right, one column past the end of the matched substring.
<int> moves pointer to the left to column int. If the pointer is already to the left of column int, the pointer is not moved.

<= moves the pointer to the left, one column past the end of the matched substring.

+int moves the pointer int columns to the right.

-int moves the pointer int columns to the left.

-L left-aligns the result of the previous item or expression in parentheses.

-R right-aligns the result of the previous item or expression in parentheses.

-C centers the result of the previous item or expression in parentheses.

*int repeats the result of the previous item or expression in parentheses int-1 times, producing a total of int copies.

Details

General Information

- When creating a pattern for matching, make the pattern as short as possible for greater efficiency. The time required for matching is roughly proportional to the length of the pattern times the length of the string that is searched.

- The algorithm used by the regular expression (RX) functions and CALL routines is a nondeterministic finite automaton.

Using Quotation Marks in Expressions

- To specify a literal that begins with a single quotation mark, use two single quotation marks instead of one.

- Literals inside a pattern must be enclosed by another layer of quotation marks. For example, "'O' 'connor'" matches an uppercase O, followed by a single quotation mark, followed by the letters "connor" in either uppercase or lowercase.

Comparisons

The regular expression (RX) functions and CALL routines work together to manipulate strings that match patterns. Use the RXPARSE function to parse a pattern you specify. Use the RXMATCH function and the CALL RXCHANGE and CALL RXSUBSTR routines to match or modify your data. Use the CALL RXFREE routine to free allocated space.

Note: Use RXPARSE only with other regular expression (RX) functions and CALL routines.

Examples

Example 1: Changing the Value of a Character in an Input String—A Basic Example

The following example searches for the character ; (semicolon) and replaces the character with a space.

data support;
  input id name $ 6--35;
datalines4;
3452 Ranklin, A.
9932 Patriot, L.;
3221 Ferraro, M.; Sandobol, S.
1228 Dietz, J.; Molina, K.

```sas
data support2 (drop=rx);
  set support;
  rx=rxparse("$';' to ' '");
  call rxchange(rx,999,name);
run;

options ls=78 ps=60 nodate pageno=1;

proc print data=support2;
run;
```

Output 4.47  Results of Replacing a Semicolon with a Space

<table>
<thead>
<tr>
<th>The SAS System</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obs</td>
<td>id</td>
</tr>
<tr>
<td>1</td>
<td>3452</td>
</tr>
<tr>
<td>2</td>
<td>9932</td>
</tr>
<tr>
<td>3</td>
<td>3221</td>
</tr>
<tr>
<td>4</td>
<td>1228</td>
</tr>
</tbody>
</table>

Example 2: Identifying and Changing the Value of Characters in an Input String  The following example uses RXPARSE to parse an input string and change the value of the string.

```sas
data test;
  input string $;
  datalines;
  abcxyzpq
  xyyzxy2x
  x2z..X7z
 ;

data _null_;  
  set;
  length to $20;
  if _n_=1 then
    rx=rxparse("' x < ? > 'z' to ABC =1 '@#%'");
  retain rx;
  drop rx;
  put string=;
  match=rxmatch(rx,string);
  put @3 match=;
  call rxsubstr(rx,string,position);
  put @3 position=;
  call rxsubstr(rx,string,position,length,score);
  put @3 position= Length= Score=;
  call rxchange(rx,999,string,to);
  put @3 to=;
```
call rxchange(rx,999,string);
run;

```r
1 data test;
2 input string $;
3 datalines;
4 NOTE: The data set WORK.TEST has 3 observations and 1 variables.
5 NOTE: DATA statement used:
6 real time 0.34 seconds
7 cpu time 0.21 seconds
8
9 data _null_
10 set;
11 length to $20;
12 if _n_=1 then
13 rx=rxparse("' x < ? > ' z to ABC =1 '@#%'";
14 retain rx;
15 drop rx;
16 put string=;
17 match=rxmatch(rx,string);
18 put $3 match=;
19 call rxsubstr(rx,string,position);
20 put $3 position=;
21 call rxsubstr(rx,string,position,length,score);
22 put $3 position= Length= Score=;
23 call rxchange(rx,999,string,to);
24 put $3 to=;
25 call rxchange(rx,999,string);
26 put $3 'New ' string=;
27 run;
```

string=abcxyzpq
match=4
position=4
length=3 score=0
to=abcabcy@#%
New string=abcabcy@

string=xyyzxyZx
match=0
position=0
length=0 score=0
to=xyyzxyZx
New string=xyyzxyZx

string=x2z..X7z
match=1
position=1
length=3 score=0
to=abc2@#%.Abc7@#%
New string=abc2@#%.

NOTE: DATA statement used:
real time 0.67 seconds
cpu time 0.45 seconds

See Also

Functions and CALL routines:
“CALL RXCHANGE Routine” on page 385
“CALL RXFREE Routine” on page 386
“CALL RXSUBSTR Routine” on page 387
“RXMATCH Function” on page 801
SAVING Function

Returns the future value of a periodic saving
Category: Financial

Syntax

SAVING(f,p,r,n)

Arguments

\( f \)

is numeric, the future amount (at the end of \( n \) periods).
Range: \( f \geq 0 \)

\( p \)

is numeric, the fixed periodic payment.
Range: \( p \geq 0 \)

\( r \)

is numeric, the periodic interest rate expressed as a decimal.
Range: \( r \geq 0 \)

\( n \)

is an integer, the number of compounding periods.
Range: \( n \geq 0 \)

Details

The SAVING function returns the missing argument in the list of four arguments from a periodic saving. The arguments are related by

\[
f = \frac{p(1 + r)\left((1 + r)^n - 1\right)}{r}
\]

One missing argument must be provided. It is then calculated from the remaining three. No adjustment is made to convert the results to round numbers.

Examples

A savings account pays a 5 percent nominal annual interest rate, compounded monthly. For a monthly deposit of $100, the number of payments that are needed to accumulate at least $12,000, can be expressed as

\[
\text{number}=\text{saving}(12000, 100, .05/12, .);
\]
The value returned is 97.18 months. The fourth argument is set to missing, which indicates that the number of payments is to be calculated. The 5 percent nominal annual rate is converted to a monthly rate of 0.05/12. The rate is the fractional (not the percentage) interest rate per compounding period.

**SCAN Function**

Selects a given word from a character expression

**Category:** Character

**Syntax**

```
SCAN(string ,n<, delimiter(s)>)
```

**Arguments**

- **string**
  - Specifies any character expression.

- **n**
  - Specifies a numeric expression that produces the number of the word in the character string you want SCAN to select. The following rules apply:
    - If \( n \) is negative, SCAN selects the word in the character string starting from the end of the string.
    - If \(|n|\) is greater than the number of words in the character string, SCAN returns a blank value.

- **delimiter**
  - Specifies a character expression that produces characters that you want SCAN to use as a word separator in the character string.

**Default:** If you omit delimiter in an ASCII environment, SAS uses the following characters:

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

In ASCII environments without the ^ character, SCAN uses the ~ character instead.

If you omit delimiter in an EBCDIC environment, SAS uses the following characters:

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

**Tip:** If you represent delimiter as a constant, enclose delimiter in quotation marks.

**Details**

If the SCAN function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

Leading delimiters before the first word in the character string do not effect SCAN. If there are two or more contiguous delimiters, SCAN treats them as one.
Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>arg='ABC.DEF(X=Y)' ; word=scan(arg,3) ; put word;</td>
<td>X=Y</td>
</tr>
<tr>
<td>word=scan(arg,-3) ; put word;</td>
<td>ABC</td>
</tr>
</tbody>
</table>

SCANQ Function

Returns the \( n \)th word from a character expression, ignoring delimiters that are enclosed in quotation marks

Category: Character

Syntax

\[
\text{SCANQ}(\text{string, } n <,\text{delimiter(s)}>)
\]

Arguments

\( \text{string} \)

specifies a character constant, variable, or expression.

\( n \)

is a numeric constant, variable, or expression that specifies the number of the word in the character string that you want SCANQ to select. The following rules apply:

- If \( n \) is positive, SCANQ counts words from left to right.
- If \( n \) is negative, SCANQ counts words from right to left.
- If \( n \) is zero, or \( |n| \) is greater than the number of words in the character string, then SCANQ returns a string that contains no characters.

\( \text{delimiter} \)

is a character constant, variable, or expression that specifies the characters that you want SCANQ to use to separate words in the character string.

Default: If you omit \( \text{delimiter} \), SCANQ uses white space characters (blank, horizontal and vertical tab, carriage return, line feed, and form feed) as delimiters.

Restriction: You cannot use single or double quotation marks as delimiters.

Details

If the SCANQ function returns a value to a variable that has not yet been assigned a length, then by default the variable is assigned a length of 200.
In the context of the SCANQ function, “word” refers to a substring that
- is bounded on the left by a delimiter or the beginning of the string
- is bounded on the right by a delimiter or the end of the string
- contains no delimiters except those that are enclosed in quotation marks.

Delimiters that are located before the first word or after the last word in the character string do not affect the SCANQ function. If two or more contiguous delimiters exist, then SCANQ treats them as one.

If the value of the character string contains quotation marks, then SCANQ ignores delimiters inside the strings in quotation marks. If the value of the character string contains unmatched quotation marks, then scanning from left to right will produce different words than scanning from right to left.

**Examples**

In this example, SCANQ reads the input string and uses a blank space as the delimiter. SCANQ ignores the delimiter that is enclosed in quotation marks.

```sas
options pageno=1 pagesize=60 linesize=80 nodate;

data nametest;
  allnames='Eleanor "Billie Holiday" Fagan';
  array new(*) $16 name1 name2 name3;
  do i=1 to dim(new);
    new(i)=scanq(allnames,i," ");
  end;
run;

proc print data=nametest;
run;
```

**Output 4.48**  Output from the SCANQ Function

<table>
<thead>
<tr>
<th>The SAS System</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obs</td>
<td>allnames</td>
</tr>
<tr>
<td>1</td>
<td>Eleanor &quot;Billie Holiday&quot; Fagan</td>
</tr>
</tbody>
</table>

**See Also**

Functions and CALL Routines:
- “SCAN Function” on page 816
- “CALL SCAN Routine” on page 388
- “CALL SCANQ Routine” on page 390
SDF Function

Computes a survival function

Category: Probability
See: “CDF Function” on page 416

Syntax

\[
\text{SDF('dist',quantile,parm-1,...,parm-k)}
\]

Arguments

'dist'
is a character string that identifies the distribution. Valid distributions are as follows:

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bernoulli</td>
<td>'BERNOULLI'</td>
</tr>
<tr>
<td>Beta</td>
<td>'BETA'</td>
</tr>
<tr>
<td>Binomial</td>
<td>'BINOMIAL'</td>
</tr>
<tr>
<td>Cauchy</td>
<td>'CAUCHY'</td>
</tr>
<tr>
<td>Chi-Square</td>
<td>'CHISQUARE'</td>
</tr>
<tr>
<td>Exponential</td>
<td>'EXPONENTIAL'</td>
</tr>
<tr>
<td>F</td>
<td>'F'</td>
</tr>
<tr>
<td>Gamma</td>
<td>'GAMMA'</td>
</tr>
<tr>
<td>Geometric</td>
<td>'GEOMETRIC'</td>
</tr>
<tr>
<td>Hypergeometric</td>
<td>'HYPERGEOMETRIC'</td>
</tr>
<tr>
<td>Laplace</td>
<td>'LAPLACE'</td>
</tr>
<tr>
<td>Logistic</td>
<td>'LOGISTIC'</td>
</tr>
<tr>
<td>Lognormal</td>
<td>'LOGNORMAL'</td>
</tr>
<tr>
<td>Negative binomial</td>
<td>'NEGBINOMIAL'</td>
</tr>
<tr>
<td>Normal</td>
<td>'NORMAL'</td>
</tr>
<tr>
<td>Normal mixture</td>
<td>'NORMALMIX'</td>
</tr>
<tr>
<td>Pareto</td>
<td>'PARETO'</td>
</tr>
<tr>
<td>Poisson</td>
<td>'POISSON'</td>
</tr>
<tr>
<td>T</td>
<td>'T'</td>
</tr>
<tr>
<td>Uniform</td>
<td>'UNIFORM'</td>
</tr>
<tr>
<td>Wald (inverse Gaussian)</td>
<td>'WALD'</td>
</tr>
<tr>
<td>Weibull</td>
<td>'WEIBULL'</td>
</tr>
</tbody>
</table>
Note: Except for T, F, and NORMALMIX, you can minimally identify any distribution by its first four characters.

quantile

is a numeric random variable.

parm-1,...,parm-k

are optional shape, location, or scale parameters appropriate for the specific distribution.

The SDF function computes the survival function (upper tail) from various continuous and discrete distributions. For more information, see the on page 417.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>y=sdf('BERN',0,.25);</td>
<td>0.25</td>
</tr>
<tr>
<td>y=sdf('BETA',0.2,3,4);</td>
<td>0.09011</td>
</tr>
<tr>
<td>y=sdf('BINOM',4,.5,10);</td>
<td>0.62305</td>
</tr>
<tr>
<td>y=sdf('CAUCHY',2);</td>
<td>0.14758</td>
</tr>
<tr>
<td>y=sdf('CHISQ',11.264,11);</td>
<td>0.42142</td>
</tr>
<tr>
<td>y=sdf('EXPO',1);</td>
<td>0.36788</td>
</tr>
<tr>
<td>y=sdf('F',3.32,2,3);</td>
<td>0.17361</td>
</tr>
<tr>
<td>y=sdf('GAMMA',1,3);</td>
<td>0.91970</td>
</tr>
<tr>
<td>y=sdf('HYPER',2,200,50,10);</td>
<td>0.47633</td>
</tr>
<tr>
<td>y=sdf('LAPLACE',1);</td>
<td>0.18394</td>
</tr>
<tr>
<td>y=sdf('LOGISTIC',1);</td>
<td>0.26894</td>
</tr>
<tr>
<td>y=sdf('LOGNORMAL',1);</td>
<td>0.5</td>
</tr>
<tr>
<td>y=sdf('NEGB',1,.5,2);</td>
<td>0.5</td>
</tr>
<tr>
<td>y=sdf('NORMAL',1.96);</td>
<td>0.025</td>
</tr>
<tr>
<td>y=pdf('NORMALMIX',2.3,3,.33,.34,.5,1.5,2.5,.79,1.6,4.3);</td>
<td>0.2819</td>
</tr>
<tr>
<td>y=sdf('PARETO',1,1);</td>
<td>1</td>
</tr>
<tr>
<td>y=sdf('POISSON',2,1);</td>
<td>0.08030</td>
</tr>
<tr>
<td>y=sdf('T',.9,5);</td>
<td>0.20469</td>
</tr>
<tr>
<td>y=sdf('UNIFORM',0.25);</td>
<td>0.75</td>
</tr>
<tr>
<td>y=sdf('WALD',1,2);</td>
<td>0.37230</td>
</tr>
<tr>
<td>y=sdf('WEIBULL',1,2);</td>
<td>0.36788</td>
</tr>
</tbody>
</table>

SECOND Function

Returns the second from a SAS time or datetime value
Category: Date and Time

Syntax
SECOND(time | datetime)

Arguments

time
specifies a SAS expression that represents a SAS time value.

datetime
specifies a SAS expression that represents a SAS datetime value.

Details
The SECOND function produces a positive, numeric value that represents a specific second of the minute. The result ranges from 0 through 59.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>time='3:19:24't; s=second(time); put s;</td>
<td>24</td>
</tr>
</tbody>
</table>

See Also

Functions:
“HOUR Function” on page 579
“MINUTE Function” on page 646

SIGN Function

Returns the sign of a value

Category: Mathematical

Syntax
SIGN(argument)
Arguments

`argument` is numeric.

Details
The SIGN function returns a value of

-1 if \( x < 0 \)
0  if \( x = 0 \)
1  if \( x > 0 \).

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x=sign(-5);</code></td>
<td>-1</td>
</tr>
<tr>
<td><code>x=sign(5);</code></td>
<td>1</td>
</tr>
<tr>
<td><code>x=sign(0);</code></td>
<td>0</td>
</tr>
</tbody>
</table>

SIN Function

Returns the sine

Category: Trigonometric

Syntax

```
SIN(argument)
```

Arguments

`argument` is numeric and is specified in radians.
**Examples**

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=sin(0.5);</td>
<td>0.4794255386</td>
</tr>
<tr>
<td>x=sin(0);</td>
<td>0</td>
</tr>
<tr>
<td>x=sin(3.14159/4);</td>
<td>.7071063121</td>
</tr>
</tbody>
</table>

**SINH Function**

Returns the hyperbolic sine  
Category: Hyperbolic

**Syntax**

SINH(argument)

**Arguments**

*argument* is numeric.

**Details**

The SINH function returns the hyperbolic sine of the argument, which is given by

\[
\left(e^{\text{argument}} - e^{-\text{argument}}\right)/2
\]

**Examples**

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=sinh(0);</td>
<td>0</td>
</tr>
<tr>
<td>x=sinh(1);</td>
<td>1.1752011936</td>
</tr>
<tr>
<td>x=sinh(-1.0);</td>
<td>-1.175201194</td>
</tr>
</tbody>
</table>

**SKEWNESS Function**

Returns the skewness
Category: Descriptive Statistics

Syntax

**SKEWNESS**(argument,argument,argument, ...)

Arguments

*argument*

is numeric. At least three nonmissing arguments are required. Otherwise, the function returns a missing value. The argument list may consist of a variable list, which is preceded by OF.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x1=skewness(0,1,1);</code></td>
<td><code>-1.732050808</code></td>
</tr>
<tr>
<td><code>x2=skewness(2,4,6,3,1);</code></td>
<td><code>0.5901286564</code></td>
</tr>
<tr>
<td><code>x3=skewness(2,0,0);</code></td>
<td><code>1.7320508076</code></td>
</tr>
<tr>
<td><code>x4=skewness(of x1-x3);</code></td>
<td><code>-0.953097714</code></td>
</tr>
</tbody>
</table>

**SLEEP Function**

Suspends the execution of a program that invokes this function for a specified period of time

Category: Special

See: SLEEP Function in the documentation for your operating environment.

Syntax

**SLEEP**(n<, unit>)

Arguments

*n*

is a numeric constant that specifies the number of units of time for which you want to suspend execution of a program.

Range: \( n \geq 0 \)

*unit*

specifies the unit of time, as a power of 10, which is applied to \( n \). For example, 1 corresponds to a second, and .001 to a millisecond.
**Details**

The SLEEP function suspends the execution of a program that invokes this function for a period of time that you specify. The program can be a DATA step, macro, IML, SCL, or anything that can invoke a function. The maximum sleep period for the SLEEP function is 46 days.

**Examples**

**Example 1: Suspending Execution for a Specified Period of Time**

The following example tells SAS to delay the execution of the DATA step PAYROLL for 20 seconds:

```sas
data payroll;
  time_slept=sleep(20,1);
  ...more SAS statements...
run;
```

**Example 2: Suspending Execution Based on a Calculation of Sleep Time**

The following example tells SAS to suspend the execution of the DATA step BUDGET until March 1, 2000, at 3:00 AM. SAS calculates the length of the suspension based on the target date and the date and time that the DATA step begins to execute.

```sas
data budget;
  sleeptime='01mar2000:03:00'dt-datetime();
  time_calc=sleep(sleeptime,1);
  ...more SAS statements...
run;
```

**See Also**

CALL routine

“CALL SLEEP Routine” on page 394

---

**SMALLEST Function**

**Returns the \( k \)th smallest nonmissing value**

**Category:** Descriptive Statistics

**Syntax**

\[
\text{SMALLEST} (k, \text{value-1}<, \text{value-2} ...>)
\]
Arguments

\( k \)

is a numeric constant, variable, or expression that specifies which value to return.

\( value \)

specifies the value of a numeric constant, variable, or expression to be processed.

Details

If \( k \) is missing, less than zero, or greater than the number of values, the result is a missing value and _ERROR_ is set to 1. Otherwise, if \( k \) is greater than the number of non-missing values, the result is a missing value but _ERROR_ is not set to 1.

Comparisons

The SMALLEST function differs from the ORDINAL function in that SMALLEST ignores missing values, but ORDINAL counts missing values.

Examples

This example compares the values that are returned by the SMALLEST function with values that are returned by the ORDINAL function.

```
options pageno=1 nodate linesize=80 pagesize=60;

data comparison;
  label smallest_num='SMALLEST Function' ordinal_num='ORDINAL Function';
  do k = 1 to 4;
    smallest_num = smallest(k, 456, 789, .Q, 123);
    ordinal_num = ordinal(k, 456, 789, .Q, 123);
    output;
  end;
run;

proc print data=comparison label noobs;
  var k smallest_num ordinal_num;
  title 'Results From the SMALLEST and the ORDINAL Functions';
run;
```

Output 4.49   Comparison of Values: The SMALLEST and the ORDINAL Functions

<table>
<thead>
<tr>
<th>Results From the SMALLEST and the ORDINAL Functions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLEST Function</td>
<td>ORDINAL Function</td>
</tr>
<tr>
<td>k</td>
<td>Function</td>
</tr>
<tr>
<td>1</td>
<td>123</td>
</tr>
<tr>
<td>2</td>
<td>456</td>
</tr>
<tr>
<td>3</td>
<td>789</td>
</tr>
<tr>
<td>4</td>
<td>.</td>
</tr>
</tbody>
</table>
See Also

Functions:
“LARGEST Function” on page 623
“ORDINAL Function” on page 692
“PCTL Function” on page 694

SOUNDEX Function

Encodes a string to facilitate searching

Category:   Character
Restriction: SOUNDEX algorithm is English-biased.

Syntax

SOUNDEX(argument)

Arguments

argument
specifies any SAS character expression.

Details

The SOUNDEX function encodes a character string according to an algorithm that was originally developed by Margaret K. Odell and Robert C. Russel (US Patents 1261167 (1918) and 1435663 (1922)). The algorithm is described in Knuth, The Art of Computer Programming, Volume 3 (See “References” on page 928). Note that the SOUNDEX algorithm is English-biased and is less useful for languages other than English.

The SOUNDEX function returns a copy of the argument that is encoded by using the following steps:

1 Retain the first letter in the argument and discard the following letters:
   A E H I O U W Y
2 Assign the following numbers to these classes of letters:
   1: B F P V
   2: C G J K Q S X Z
   3: D T
   4: L
   5: M N
   6: R
3 If two or more adjacent letters have the same classification from Step 2, then discard all but the first. (Adjacent refers to the position in the word prior to discarding letters.)
The algorithm that is described in Knuth adds trailing zeros and truncates the result to the length of 4. You can perform these operations with other SAS functions.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=soundex('Paul');</td>
<td>P4</td>
</tr>
<tr>
<td>put x;</td>
<td></td>
</tr>
<tr>
<td>word='amnesty';</td>
<td>A523</td>
</tr>
<tr>
<td>x=soundex(word);</td>
<td></td>
</tr>
<tr>
<td>put x;</td>
<td></td>
</tr>
</tbody>
</table>

SPEDIS Function

Determines the likelihood of two words matching, expressed as the asymmetric spelling distance between the two words

Category: Character

Syntax

SPEDIS(query,keyword)

Arguments

query

Identifies the word to query for the likelihood of a match. SPEDIS removes trailing blanks before comparing the value.

keyword

Specifies a target word for the query. SPEDIS removes trailing blanks before comparing the value.

Details

SPEDIS returns the distance between the query and a keyword, a nonnegative value that is usually less than 100 but never greater than 200 with the default costs.

SPEDIS computes an asymmetric spelling distance between two words as the normalized cost for converting the keyword to the query word by using a sequence of operations. SPEDIS(QUERY, KEYWORD) is not the same as SPEDIS(KEYWORD, QUERY).

Costs for each operation that is required to convert the keyword to the query are
The distance is the sum of the costs divided by the length of the query. If this ratio is greater than one, the result is rounded down to the nearest whole number.

Comparisons

The SPEDIS function is similar to the COMPLEV and COMPGED functions, but COMPLEV and COMPGED are much faster, especially for long strings.

Examples

```plaintext
options nodate pageno=1 linesize=64;

data words;
  input Operation $ Query $ Keyword $;
  Distance = spedis(query,keyword);
  Cost = distance * length(query);
datalines;
  match  fuzzy  fuzzy
  singlet fuzy  fuzzy
  doublet fuuzzy fuzzy
  swap    fzuy  fuzzy
  truncate fuzzz fuzzy
  append  fuzys  fuzzy
  delete  fzzy  fuzzy
  insert  fluzzy fuzzy
  replace fizzy  fuzzy
  firstdel uzzy  fuzzy
  firstins pfuzzy fuzzy
  firstrep wuzzy fuzzy
  several floozy fuzzy
;```
proc print data = words;
run;

The output from the DATA step is as follows.

### Output 4.50  Costs for SPEDIS Operations

<table>
<thead>
<tr>
<th>Obs</th>
<th>Operation</th>
<th>Query</th>
<th>Keyword</th>
<th>Distance</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>match</td>
<td>fuzzy</td>
<td>fuzzy</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>singlet</td>
<td>fuzzy</td>
<td>fuzzy</td>
<td>6</td>
<td>24</td>
</tr>
<tr>
<td>3</td>
<td>doublet</td>
<td>fuzzys</td>
<td>fuzzy</td>
<td>8</td>
<td>48</td>
</tr>
<tr>
<td>4</td>
<td>swap</td>
<td>fuzz</td>
<td>fuzzy</td>
<td>10</td>
<td>50</td>
</tr>
<tr>
<td>5</td>
<td>truncate</td>
<td>fuzz</td>
<td>fuzzy</td>
<td>12</td>
<td>48</td>
</tr>
<tr>
<td>6</td>
<td>append</td>
<td>fuzzys</td>
<td>fuzzy</td>
<td>5</td>
<td>30</td>
</tr>
<tr>
<td>7</td>
<td>delete</td>
<td>fuzzy</td>
<td>fuzzy</td>
<td>12</td>
<td>48</td>
</tr>
<tr>
<td>8</td>
<td>insert</td>
<td>fluzzy</td>
<td>fuzzy</td>
<td>16</td>
<td>96</td>
</tr>
<tr>
<td>9</td>
<td>replace</td>
<td>fizzy</td>
<td>fuzzy</td>
<td>20</td>
<td>100</td>
</tr>
<tr>
<td>10</td>
<td>firstdel</td>
<td>uuzzy</td>
<td>fuzzy</td>
<td>25</td>
<td>100</td>
</tr>
<tr>
<td>11</td>
<td>firstins</td>
<td>pfuzzy</td>
<td>fuzzy</td>
<td>33</td>
<td>198</td>
</tr>
<tr>
<td>12</td>
<td>firstrep</td>
<td>wuzzy</td>
<td>fuzzy</td>
<td>40</td>
<td>200</td>
</tr>
<tr>
<td>13</td>
<td>several</td>
<td>floozy</td>
<td>fuzzy</td>
<td>50</td>
<td>300</td>
</tr>
</tbody>
</table>

### See Also

Functions:

- “COMPLEV Function” on page 452
- “COMPGED Function” on page 447

---

**SQRT Function**

**Returns the square root of a value**

**Category:** Mathematical

**Syntax**

\[ \text{SQRT}(\text{argument}) \]

**Arguments**

\[ \text{argument} \]

is numeric and must be nonnegative.
Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=sqrt(36);</td>
<td>6</td>
</tr>
<tr>
<td>x=sqrt(25);</td>
<td>5</td>
</tr>
<tr>
<td>x=sqrt(4.4);</td>
<td>2.0976176963</td>
</tr>
</tbody>
</table>

STD Function

Returns the standard deviation

Category: Descriptive Statistics

Syntax

STD(argument,argument,...)

Arguments

argument is numeric. At least two nonmissing arguments are required. Otherwise, the function returns a missing value. The argument list can consist of a variable list, which is preceded by OF.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1=std(2,6);</td>
<td>2.8284271247</td>
</tr>
<tr>
<td>x2=std(2,6,..);</td>
<td>2.8284271427</td>
</tr>
<tr>
<td>x3=std(2,4,6,3,1);</td>
<td>1.9235384062</td>
</tr>
<tr>
<td>x4=std(of x1-x3);</td>
<td>0.5224377453</td>
</tr>
</tbody>
</table>

STDERR Function

Returns the standard error of the mean
Category: Descriptive Statistics

Syntax

STDERR(argument,argument,...)

Arguments

argument
is numeric. At least two nonmissing arguments are required. Otherwise, the function returns a missing value. The argument list can consist of a variable list, which is preceded by OF.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1=stderr(2,6);</td>
<td>2</td>
</tr>
<tr>
<td>x2=stderr(2,6, .);</td>
<td>2</td>
</tr>
<tr>
<td>x3=stderr(2,4,6,3,1);</td>
<td>0.8602325267</td>
</tr>
<tr>
<td>x4=stderr(of x1-x3);</td>
<td>0.3799224911</td>
</tr>
</tbody>
</table>

STFIPS Function

Converts state postal codes to FIPS state codes

Category: State and ZIP Code

Syntax

STFIPS(postal-code)

Arguments

postal-code
specifies a character expression that contains the two-character standard state postal code. Characters can be mixed case. The function ignores trailing blanks, but generates an error if the expression contains leading blanks.

Details

The STFIPS function converts a two-character state postal code (or world-wide GSA geographic code for U.S. territories) to the corresponding numeric U.S. Federal Information Processing Standards (FIPS) code.
Comparisons

The STFIPS, STNAME, and STNAMEL functions take the same argument but return different values. STFIPS returns a numeric U.S. Federal Information Processing Standards (FIPS) code. STNAME returns an uppercase state name. STNAMEL returns a mixed case state name.

Examples

The examples show the differences when using STFIPS, STNAME, and STNAMEL.

```
<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>fips=stfips('NC');</td>
<td>37</td>
</tr>
<tr>
<td>put fips;</td>
<td></td>
</tr>
<tr>
<td>state=stname('NC');</td>
<td>NORTH CAROLINA</td>
</tr>
<tr>
<td>put state;</td>
<td></td>
</tr>
<tr>
<td>state=stnamel('NC');</td>
<td>North Carolina</td>
</tr>
<tr>
<td>put state;</td>
<td></td>
</tr>
</tbody>
</table>
```

See Also

Functions:
- “FIPNAME Function” on page 540
- “FIPNAMEL Function” on page 541
- “FIPSTATE Function” on page 542
- “STNAME Function” on page 833,
- “STNAMEL Function” on page 834

STNAME Function

Converts state postal codes to uppercase state names

Category: State and ZIP Code

Syntax

```
STNAME(postal-code)
```

Arguments

`postal-code`

specifies a character expression that contains the two-character standard state postal code. Characters can be mixed case. The function ignores trailing blanks, but generates an error if the expression contains leading blanks.
Details

The STNAME function converts a two-character state postal code (or world-wide GSA geographic code for U.S. territories) to the corresponding state name in uppercase.

*Note:* For Version 6, the maximum length of the value that is returned is 200 characters. For Version 7 and beyond, the maximum length is 20 characters.

Comparisons

The STFIPS, STNAME, and STNAMEL functions take the same argument but return different values. STFIPS returns a numeric U.S. Federal Information Processing Standards (FIPS) code. STNAME returns an uppercase state name. STNAMEL returns a mixed case state name.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>fips=stfips ('NC'); put fips;</td>
<td>37</td>
</tr>
<tr>
<td>state=stname('NC'); put state;</td>
<td>NORTH CAROLINA</td>
</tr>
<tr>
<td>state=stnamel('NC'); put state;</td>
<td>North Carolina</td>
</tr>
</tbody>
</table>

See Also

Functions:

“FIPNAME Function” on page 540
“FIPNAMEL Function” on page 541
“FIPSTATE Function” on page 542
“STFIPS Function” on page 832
“STNAMEL Function” on page 834

STNAMEL Function

Converts state postal codes to mixed case state names

Category: State and ZIP Code

Syntax

```
STNAMEL(postal-code)
```
Arguments

**postal-code**

specifies a character expression that contains the two-character standard state postal code. Characters can be mixed case. The function ignores trailing blanks, but generates an error if the expression contains leading blanks.

Details

If the STNAMEL function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 20.

The STNAMEL function converts a two-character state postal code (or world-wide GSA geographic code for U.S. territories) to the corresponding state name in mixed case.

**Note:** For Version 6, the maximum length of the value that is returned is 200 characters. For Version 7 and beyond, the maximum length is 20 characters.

Comparisons

The STFIPS, STNAME, and STNAMEL functions take the same argument but return different values. STFIPS returns a numeric U.S. Federal Information Processing Standards (FIPS) code. STNAME returns an uppercase state name. STNAMEL returns a mixed case state name.

Examples

The examples show the differences when using STFIPS, STNAME, and STNAMEL.

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
</table>
| fips=stfips ('NC');  
put fips; | 37 |
| state=stname('NC');  
put state; | NORTH CAROLINA |
| state=stnamel('NC');  
put state; | North Carolina |

See Also

Functions:

“FIPNAME Function” on page 540  
“FIPNAMEL Function” on page 541  
“FIPSTATE Function” on page 542  
“STFIPS Function” on page 832

### STRIP Function

Returns a character string with all leading and trailing blanks removed
Category: Character

Syntax

\texttt{STRIP(string)}

Arguments

\textit{string}

is a character constant, variable, or expression.

Details

If the STRIP function returns a value to a variable that has not yet been assigned a length, by default the variable length is determined by the length of the first argument. The STRIP function returns the argument with all leading and trailing blanks removed. If the argument is blank, STRIP returns a string with a length of zero.

Assigning the results of STRIP to a variable does not affect the length of the receiving variable. If the value that is trimmed is shorter than the length of the receiving variable, SAS pads the value with new trailing blanks.

\textit{Note:} The STRIP function is useful for concatenation because the concatenation operator does not remove trailing blanks. △

Comparisons

The following list compares the STRIP function with the TRIM and TRIMN functions:

- For strings that are blank, the STRIP and TRIMN functions return a string with a length of zero, whereas the TRIM function returns a single blank.
- For strings that lack leading blanks, the STRIP and TRIMN functions return the same value.
- For strings that lack leading blanks but have at least one non-blank character, the STRIP and TRIM functions return the same value.

\textit{Note:} \texttt{STRIP(string)} returns the same result as \texttt{TRIMN(LEFT(string))}, but the STRIP function runs faster. △

Examples

The following example shows the results of using the STRIP function to delete leading and trailing blanks.

```sql
options pageno=1 nodate ls=80 ps=60;

data lengthn;
  input string $char8.;
  original = '*' || string || '*';
  stripped = '*' || strip(string) || '*';
  datalines;
  abcd
  abcd
```

```sql
```
abcd
abcdefgh
xyz;

proc print data=lengthn;
run;

Output 4.51 Results from the STRIP Function

<table>
<thead>
<tr>
<th>Obs</th>
<th>string</th>
<th>original</th>
<th>stripped</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>abcd</td>
<td>*abcd</td>
<td><em>abcd</em></td>
</tr>
<tr>
<td>2</td>
<td>abcd</td>
<td>* abcd</td>
<td><em>abcd</em></td>
</tr>
<tr>
<td>3</td>
<td>abcd</td>
<td>* abcd*</td>
<td><em>abcd</em></td>
</tr>
<tr>
<td>4</td>
<td>abcd</td>
<td>* abcd*</td>
<td><em>abcd</em></td>
</tr>
<tr>
<td>5</td>
<td>xyz</td>
<td>*xyz</td>
<td><em>xyz</em></td>
</tr>
</tbody>
</table>

See Also

Functions:
“CAT Function” on page 408
“CATS Function” on page 410
“CATT Function” on page 412
“CATX Function” on page 414
“LEFT Function” on page 625
“TRIM Function” on page 863
“TRIMN Function” on page 865

SUBPAD Function

Returns a substring that has a length you specify, using blank padding if necessary

Category: Character

Syntax

SUBPAD(string, position <, length>)

Arguments

string
specifies a character string.
position
is a positive integer that specifies the position of the first character in the substring.

length
is a non-negative integer that specifies the length of the substring. If you do not specify length, the SUBPAD function returns the substring that extends from the position that you specify to the end of the string.

Details
If the SUBPAD function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.
If the substring that you specify extends beyond the length of the string, the result is padded with blanks.

Comparisons
The SUBPAD function is similar to the SUBSTR function except for the following differences:

- If the value of length in SUBPAD is zero, SUBPAD returns a zero-length string. If the value of length in SUBSTR is zero, SUBSTR
  - writes a note to the log stating that the third argument is invalid
  - sets _ERROR_.=1
  - returns the substring that extends from the position that you specified to the end of the string.

- If the substring that you specify extends past the end of the string, SUBPAD pads the result with blanks to yield the length that you requested. If the substring that you specify extends past the end of the string, SUBSTR
  - writes a note to the log stating that the third argument is invalid
  - sets _ERROR_.=1
  - returns the substring that extends from the position that you specified to the end of the string.

See Also

Function:
“SUBSTRN Function” on page 841

---

SUBSTR (left of =) Function

Replaces character value contents

Category: Character

Syntax

SUBSTR(variable, position <, length >)=characters-to-replace
Arguments

variable
specifies a character variable.

position
specifies a numeric expression that is the beginning character position.

length
specifies a numeric expression that is the length of the substring that will be replaced.
Restriction: length cannot be larger than the length of the expression that remains in variable after position.
Tip: If you omit length, SAS uses all of the characters on the right side of the assignment statement to replace the values of variable.

characters-to-replace
specifies a character expression that will replace the contents of variable.
Tip: Enclose a literal string of characters in quotation marks.

Details
If you use an undeclared variable, it will be assigned a default length of 8 when the SUBSTR function is compiled.
When you use the SUBSTR function on the left side of an assignment statement, SAS replaces the value of variable with the expression on the right side. SUBSTR replaces length characters starting at the character that you specify in position.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a='KIDNAP';</td>
<td></td>
</tr>
<tr>
<td>substr(a,1,3)=’CAT’;</td>
<td></td>
</tr>
<tr>
<td>put a;</td>
<td>CATNAP</td>
</tr>
<tr>
<td>b=a;</td>
<td></td>
</tr>
<tr>
<td>substr(b,4)=’TY’;</td>
<td></td>
</tr>
<tr>
<td>put b;</td>
<td>CATTY</td>
</tr>
</tbody>
</table>

See Also

Function:
“SUBSTR (right of =) Function” on page 839

SUBSTR (right of =) Function

Extracts a substring from an argument
Category: Character

Syntax

<variable=>SUBSTR(string, position<,length>)

Arguments

variable
specifies a valid SAS variable name.

string
specifies any SAS character expression.

position
specifies a numeric expression that is the beginning character position.

length
specifies a numeric expression that is the length of the substring to extract.

Interaction: If length is zero, a negative value, or larger than the length of the expression that remains in string after position, SAS extracts the remainder of the expression. SAS also sets _ERROR_ to 1 and prints a note to the log indicating that the length argument is invalid.

Tip: If you omit length, SAS extracts the remainder of the expression.

Details

If the SUBSTR function returns a value to a variable that has not yet been assigned a length, by default the variable length is determined by the length of the first argument. The SUBSTR function returns a portion of an expression that you specify in string. The portion begins with the character that you specify by position, and is the number of characters that you specify in length.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>date='06MAY98';</td>
<td></td>
</tr>
<tr>
<td>month=substr(date,3,3);</td>
<td>MAY 98</td>
</tr>
<tr>
<td>year=substr(date,6,2);</td>
<td></td>
</tr>
<tr>
<td>put @1 month @5 year;</td>
<td></td>
</tr>
</tbody>
</table>

See Also

Functions:

“SUBPAD Function” on page 837
SUBSTRN Function

Returns a substring, allowing a result with a length of zero

Category: Character

Syntax

SUBSTRN(string, position <, length>)

Arguments

string
specifies a character string.

position
is an integer that specifies the position of the first character in the substring.

length
is an integer that specifies the length of the substring. If you do not specify length, the SUBSTRN function returns the substring that extends from the position that you specify to the end of the string.

Details

If the SUBSTRN function returns a value to a variable that has not yet been assigned a length, by default the variable length is determined by the length of the first argument.

The following information applies to the SUBSTRN function:

- The SUBSTRN function returns a string with a length of zero if either position or length has a missing value.
- If the position that you specify is non-positive, the result is truncated at the beginning, so that the first character of the result is the first character of the string. The length of the result is reduced accordingly.
- If the length that you specify extends beyond the end of the string, the result is truncated at the end, so that the last character of the result is the last character of the string.

Comparisons

The following table lists comparisons between the SUBSTRN and the SUBSTR functions.
Table 4.5  Comparisons between SUBSTRN and SUBSTR

<table>
<thead>
<tr>
<th>If ...</th>
<th>in the function ...</th>
<th>the function ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>the value of position is non-positive</td>
<td>SUBSTRN</td>
<td>returns a result beginning at the first character of the string.</td>
</tr>
</tbody>
</table>
| the value of position is non-positive | SUBSTR | □ writes a note to the log stating that the second argument is invalid  
| | | □ sets _ERROR_ =1  
| | | □ returns the substring that extends from the position that you specified to the end of the string. |
| the value of length is non-positive | SUBSTRN | returns a result with a length of zero. |
| the value of length is non-positive | SUBSTR | □ writes a note to the log stating that the third argument is invalid  
| | | □ sets _ERROR_ =1  
| | | □ returns the substring that extends from the position that you specified to the end of the string. |
| the substring that you specify extends past the end of the string | SUBSTRN | truncates the result. |
| the substring that you specify extends past the end of the string | SUBSTR | □ writes a note to the log stating that the third argument is invalid  
| | | □ sets _ERROR_ =1  
| | | □ returns the substring that extends from the position that you specified to the end of the string. |

Examples

The following example shows how the SUBSTRN function works with strings.

```plaintext
options pageno=1 nodate ls=80 ps=60;

data test;
  retain string "abcd";
  drop string;
  do Position = -1 to 6;
    do Length = max(-1,-position) to 7-position;
      Result = substrn(string, position, length);
      output;
  end;
```

end;
datalines;
abcd;

proc print noobs data=test;
run;

Output 4.52  Results from the SUBSTRN Function

<table>
<thead>
<tr>
<th>Position</th>
<th>Length</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>-1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>3</td>
<td>ab</td>
</tr>
<tr>
<td>-1</td>
<td>4</td>
<td>abc</td>
</tr>
<tr>
<td>-1</td>
<td>5</td>
<td>abcd</td>
</tr>
<tr>
<td>-1</td>
<td>6</td>
<td>abcd</td>
</tr>
<tr>
<td>-1</td>
<td>7</td>
<td>abcd</td>
</tr>
<tr>
<td>-1</td>
<td>8</td>
<td>abcd</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>a</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>ab</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>abc</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>abcd</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>abcd</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
<td>abcd</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>ab</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>abc</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>abcd</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>abcd</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>abcd</td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>bc</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>bcd</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>bcd</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>bcd</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>cd</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>cd</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>cd</td>
</tr>
<tr>
<td>4</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>d</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>d</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>d</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>d</td>
</tr>
<tr>
<td>5</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
SUM Function

Returns the sum of the nonmissing arguments

Category: Descriptive Statistics

Syntax

SUM(argument, argument, ...)

Arguments

argument

is numeric. If all the arguments have missing values, the result is a missing value.

The argument list can consist of a variable list, which is preceded by OF.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1=sum(4,9,3,8);</td>
<td>24</td>
</tr>
<tr>
<td>x2=sum(4,9,3,8,..);</td>
<td>24</td>
</tr>
<tr>
<td>x1=9; x2=39; x3=sum(of x1-x2);</td>
<td>48</td>
</tr>
<tr>
<td>x1=5; x2=6; x3=4; x4=9;</td>
<td></td>
</tr>
<tr>
<td>y1=34; y2=12; y3=74; y4=39;</td>
<td></td>
</tr>
<tr>
<td>result=sum(of x1-x4, of y1-y5);</td>
<td>183</td>
</tr>
<tr>
<td>x1=55; x2=35; x3=6; x4=sum(of x1-x3, 5);</td>
<td>101</td>
</tr>
<tr>
<td>x1=7; x2=7; x5=sum(x1-x2);</td>
<td>0</td>
</tr>
<tr>
<td>y1=20; y2=30; x6=sum(of y:);</td>
<td>50</td>
</tr>
</tbody>
</table>
SYMEXIST Function

Returns an indication of the existence of a macro variable

Category: Macro

See: SYMEXIST Function in SAS Macro Language: Reference

Syntax

SYMEXIST (argument)

Argument

argument

can be one of the following items:

- the name of a macro variable within double quotation marks but without an ampersand.
- the name of a DATA step character variable, specified with no quotation marks, which contains a macro variable name.
- a character expression that constructs a macro variable name.

Details

The SYMEXIST function searches any enclosing local symbol tables and then the global symbol table for the indicated macro variable and returns 1 if the macro variable is found or 0 if the macro variable is not found.

For more information, see the “SYMEXIST Function” in SAS Macro Language: Reference.

SYMGET Function

Returns the value of a macro variable during DATA step execution

Category: Macro

Syntax

SYMGET(argument)

Arguments

argument

is a character expression that identifies the macro variable whose value you want to retrieve.
Details

If the SYMGET function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

The SYMGET function returns the value of a macro variable during DATA step execution. For more information, see the “SYMGET Function” in SAS Macro Language: Reference.

See Also

CALL routine:

“CALL SYMPUT Routine” on page 401

SAS Macro Language: Reference

SYMGLOBL Function

Returns an indication as to whether a macro variable is in a global scope to the DATA step during DATA step execution.

Category: Macro

See: SYMGLOBL Function in SAS Macro Language: Reference

Syntax

SYMGLOBL (argument)

Argument

argument
can be one of the following items:

- the name of a macro variable within double quotation marks but without an ampersand.
- the name of a DATA step character variable, specified with no quotation marks, which contains a macro variable name.
- a character expression that constructs a macro variable name.

Details

The SYMGLOBL function searches only the global symbol table for the indicated macro variable and returns 1 if the macro variable is found or 0 if the macro variable is not found.

SYMGLOBL is fully documented in SAS Macro Language: Reference.
SYMLOCAL Function

Returns an indication as to whether a macro variable is in a local scope to the DATA step during DATA step execution

Category: Macro
See: SYMLOCAL Function in SAS Macro Language: Reference

Syntax

SYMLOCAL (argument)

Argument

argument
can be one of the following items:
- the name of a macro variable within double quotation marks but without an ampersand.
- the name of a DATA step character variable, specified with no quotation marks, which contains a macro variable name.
- a character expression that constructs a macro variable name.

Details

The SYMLOCAL function searches the enclosing local symbol tables for the indicated macro variable and returns 1 if the macro variable is found or 0 if the macro variable is not found.

SYMLOCAL is fully documented in SAS Macro Language: Reference.

SYSGET Function

Returns the value of the specified operating environment variable

Category: Special
See: SYSGET Function in the documentation for your operating environment.

Syntax

SYSGET(operating-environment-variable)
Arguments

operating-environment-variable

is the name of an operating environment variable. The case of
operating-environment-variable must agree with the case that is stored in the
operating environment. Trailing blanks in the argument of SYSGET are significant.
Use the TRIM function to remove them.

Operating Environment Information: The term operating-environment-variable used in
the description of this function refers to a name that represents a numeric, character, or
logical value in the operating environment. Refer to the SAS documentation for your
operating environment for details.

Details

If the SYSGET function returns a value to a variable that has not yet been assigned a
length, by default the variable is assigned a length of 200.

If the value of the operating environment variable is truncated or the variable is not
defined in the operating environment, SYSGET displays a warning message in the SAS
log.

Examples

This example obtains the value of two environment variables in the UNIX
environment:

data _null_;  
  length result $200;  
  input env_var $;  
  result=sysget(trim(env_var));  
  put env_var= result=;  
datalines;
USER
PATH
;

Executing this DATA step for user ABCDEF displays these lines:

  ENV_VAR=USER RESULT=abcdef
  ENV_VAR=PATH RESULT=path-for-abcdef

SYSMSG Function

Returns the text of error messages or warning messages from the last data set or external file
function execution

Category: SAS File I/O
Category: External Files
**Syntax**

SYSMSG()

**Details**

SYSMSG returns the text of error messages or warning messages that are produced when a data set or external file access function encounters an error condition. If no error message is available, the returned value is blank. The internally stored error message is reset to blank after a call to SYSMSG, so subsequent calls to SYSMSG before another error condition occurs return blank values.

**Example**

This example uses SYSMSG to write to the SAS log the error message generated if FETCH cannot copy the next observation into the Data Set Data Vector. The return code is 0 only when a record is fetched successfully:

```sas
%let rc=%sysfunc(fetch(&dsid));
%if &rc ne 0 %then
  %put %sysfunc(sysmsg());
```

**See Also**

Functions:

- “FETCH Function” on page 522
- “SYSRC Function” on page 853

---

**SYSPARM Function**

Returns the system parameter string

**Category:** Special

**Syntax**

SYSPARM()

**Details**

If the SYSPARM function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

SYSPARM allows you to access a character string specified with the SYSPARM= system option at SAS invocation or in an OPTIONS statement.

**Note:** If the SYSPARM= system option is not specified, the SYSPARM function returns a null string. △
Example

This example shows the SYSPARM= system option and the SYSPARM function.

```sas
options sysparm='yes';
data a;
  if sysparm()='yes' then
    do;
      ...SAS Statements...
    end;
run;
```

See Also

System option:

“SYSPARM= System Option” on page 1610

---

**SYSPROCESSID Function**

**Returns the process id of the current process**

Category: Special

**Syntax**

```sas
SYSPROCESSID()
```

**Details**

The SYSPROCESSID function returns the 32–character hexadecimal id of the current process. This id can be passed to the SYSPROCESSNAME function to obtain the name of the current process.

**Examples**

**Example 1: Using a Data Step**

The following DATA step writes the current process id to the SAS log:

```sas
data _null_;  
id=sysprocessid();  
put id;  
run;
```

**Example 2: Using SAS Macro Language**

The following SAS Macro Language code writes the current process id to the SAS log:

```sas
%let id=%sysfunc(sysprocessid());
%put &id;
```
SYSPROCESSNAME Function

Returns the process name associated with a given process id or the name of the current process

Category: Special

Syntax

SYSPROCESSNAME(<process_id>)

Arguments

process_id

specifies a 32–character hexadecimal process id.

Details

The SYSPROCESSNAME function returns the process name associated with the process id you supply as an argument. You can use the value returned from the SYSPROCESSID function as the argument to SYSPROCESSNAME. If you omit the argument, then SYSPROCESSNAME returns the name of the current process.

You can also use the values stored in the automatic macro variables SYSPROCESSID and SYSSTARTID as arguments to SYSPROCESSNAME.

Examples

Example 1: Using SYSPROCESSNAME Without an Argument in a Data Step

The following DATA step writes the current process name to the SAS log:

```
data _null_;  
  name=syssize();  
  put name;  
run;  
```

Example 2: Using SYSPROCESSNAME With an Argument in SAS Macro Language

The following SAS Macro Language code writes the process name associated with the given process id to the SAS log:

```
%let id=&sysprocessid;  
%let name=%sysfunc(syssize(&id));  
%put &name;  
```
SYSPROD Function

Determines if a product is licensed

Category:  Special

Syntax

SYSPROD(product-name)

Arguments

product-name
specifies a character expression that resolves to the name of a SAS product.

Details

The SYSPROD function returns 1 if a specific SAS software product is licensed, 0 if it is a SAS software product but not licensed for your system, and -1 if the product name is not recognized. Use SYSPROD in the DATA step, in an IML step, or in an SCL program.

If SYSPROD indicates that a product is licensed, it means that the final license expiration date has not passed. Use the SETINIT procedure to determine the final expiration date for the product.

It is possible for a SAS software product to exist on your system even though the product is no longer licensed. However, SAS cannot access this product.

You can enter the product name in uppercase, in lowercase, or in mixed case. You can prefix the product with 'SAS/'. You can prefix SAS/ACCESS product names with 'ACC-'. Use the SETINIT procedure to obtain a list of products available on your system.

Examples

These examples determine if a specified product is licensed:

- x=sysprod('graph');
  If SAS/GRAPH software is currently licensed, then SYSPROD returns a value of 1. If SAS/GRAPH software is not currently licensed, then SYSPROD returns a value of 0.
- x=sysprod('abc');
  SYSPROD returns a value of -1 because ABC is not a valid product name.
- x=sysprod('base');
or
x=sysprod('base sas');
SYSPROD always returns a value of 1 because the Base product must be licensed for the DATA step to run successfully.

## SYSRC Function

**Returns a system error number**

**Category:** SAS File I/O  
**Category:** External Files

### Syntax

```plaintext
SYSRC()
```

### Details

SYSRC returns the error number for the last system error encountered by a call to one of the data set functions or external file functions.

### Example

This example determines the error message if FILEREF does not exist:

```plaintext
%if %sysfunc(fileref(myfile)) ne 0 %then
  %put %sysfunc(sysrc()) - %sysfunc(sysmsg());
```

### See Also

Functions:
- “FILEREF Function” on page 531
- “SYMSM Function” on page 848

## SYSTEM Function

**Issues an operating environment command during a SAS session and returns the system return code**

**Category:** Special  
See: SYSTEM Function in the documentation for your operating environment.

### Syntax

```plaintext
SYSTEM(command)
```
**Arguments**

**command**

specifies any of the following: a system command that is enclosed in quotation marks (explicit character string), an expression whose value is a system command, or the name of a character variable whose value is a system command that is executed.

**Operating Environment Information:** See the SAS documentation for your operating environment for information on what you can specify. The system return code is dependent on your operating environment.

**Restriction:** The length of the command cannot be greater than 1024 characters, including trailing blanks.

**Comparisons**

The SYSTEM function is similar to the X statement, 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, and accepts expressions as arguments. The X statement is a global statement and executes as a DATA step is being compiled, regardless of whether SAS encounters a conditional statement.

**Example**

Execute the host command TIMEDATA if the macro variable SYSDAY is **Friday**.

```sas
data _null_;  
  if "&sysday"="Friday" then do;  
    rc=system("timedata");  
  end;  
  else rc=system("errorck");  
run;
```

**See Also**

CALL Routine:

“CALL SYSTEM Routine” on page 403

Statement:

“X Statement” on page 1437

---

**TAN Function**

**Returns the tangent**

**Category:** Trigonometric

**Syntax**

TAN(argument)
Arguments

*argument* is numeric and is specified in radians.

**Restriction:** cannot be an odd multiple of $\pi/2$

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=tan(0.5);</td>
<td>0.5463024898</td>
</tr>
<tr>
<td>x=tan(0);</td>
<td>0</td>
</tr>
<tr>
<td>x=tan(3.14159/3);</td>
<td>1.7320472695</td>
</tr>
</tbody>
</table>

**TANH Function**

Returns the hyperbolic tangent

**Category:** Hyperbolic

**Syntax**

`TANH(argument)`

**Arguments**

*argument* is numeric.

**Details**

The TANH function returns the hyperbolic tangent of the argument, which is given by

$$\frac{e^{argument} - e^{-argument}}{e^{argument} + e^{-argument}}$$
Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=tanh(0);</td>
<td>0</td>
</tr>
<tr>
<td>x=tanh(0.5);</td>
<td>0.4621171573</td>
</tr>
<tr>
<td>x=tanh(-0.5);</td>
<td>-0.462117157</td>
</tr>
</tbody>
</table>

TIME Function

Returns the current time of day

Category: Date and Time

Syntax

TIME()

Example

SAS assigns CURRENT a SAS time value corresponding to 14:32:00 if the following statements are executed exactly at 2:32 PM:

```sas
    current=time();
    put current=time.;
```

TIMEPART Function

Extracts a time value from a SAS datetime value

Category: Date and Time

Syntax

TIMEPART(datetime)

Arguments

datetime

specifies a SAS expression that represents a SAS datetime value.
Example

SAS assigns TIME a SAS value that corresponds to 10:40:17 if the following statements are executed exactly at 10:40:17 AM on any date:

```sas
datim=datetime();
time=timepart(datim);
```

---

**TINV Function**

Returns a quantile from the $t$ distribution

**Category:** Quantile

**Syntax**

`TINV(p,df,nc)`

**Arguments**

$p$

- is a numeric probability.
  - Range: $0 < p < 1$

$df$

- is a numeric degrees of freedom parameter.
  - Range: $df > 0$

$nc$

- is an optional numeric noncentrality parameter.

**Details**

The TINV function returns the $p^{th}$ quantile from the Student’s $t$ distribution with degrees of freedom $df$ and a noncentrality parameter $nc$. The probability that an observation from a $t$ distribution is less than or equal to the returned quantile is $p$.

`TINV` accepts a noninteger degree of freedom parameter $df$. If the optional parameter $nc$ is not specified or is 0, the quantile from the central $t$ distribution is returned.

**CAUTION:**

For large values of $nc$, the algorithm can fail; in that case, a missing value is returned. △

**Note:** TINV is the inverse of the PROBT function. △

**Examples**

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x=tinv(.95,2);</code></td>
<td>2.9199855804</td>
</tr>
<tr>
<td><code>x=tinv(.95,2.5,3);</code></td>
<td>1.033833625</td>
</tr>
</tbody>
</table>
**TNONCT Function**

Returns the value of the noncentrality parameter from the student's $t$ distribution

**Category:** Mathematical

**Syntax**

`TNONCT(x, df, prob)`

**Arguments**

$x$

is a numeric random variable.

$df$

is a numeric degrees-of-freedom parameter.

Range: $df > 0$

$prob$

is a probability.

Range: $0 < prob < 1$

**Details**

The TNONCT function returns the nonnegative noncentrality parameter from a noncentral $t$ distribution whose parameters are $x$, $df$, and $nc$. A Newton-type algorithm is used to find a root $nc$ of the equation

$$P_t(x | df, nc) - prob = 0$$

where

$$P_t(x | df, nc) = \frac{1}{\Gamma(\frac{df}{2})} \int_0^{\infty} v^{\frac{df}{2} - 1} e^{-v} \int_{-\infty}^{\frac{x}{\sqrt{v}}} e^{-\frac{(x-nc)^2}{v}} du dv$$

If the algorithm fails to converge to a fixed point, a missing value is returned.

**Example**

```plaintext
data work;
  x=2;
  df=4;
  do nc=1 to 3 by .5;
    prob=probt(x, df, nc);
    ncc=tnonct(x, df, prob);
```

Functions and CALL Routines

TRANSLATE Function

Replaces specific characters in a character expression

TODAY Function

Returns the current date as a SAS date value

Category: Date and Time

Alias: DATE

Syntax

TODAY()

Details

The TODAY function produces the current date in the form of a SAS date value, which is the number of days since January 1, 1960.

Example

These statements illustrate a practical use of the TODAY function:

```sas
data _null_;  
tday=today();  
if (tday-datedue)> 15 then  
do;  
   put 'As of ' tday date9. ' Account #’  
      account ‘is more than 15 days overdue.’;  
   end;  
run;
```

Output 4.53  Computations of the Noncentrality Parameter from the t Distribution

<table>
<thead>
<tr>
<th>OBS</th>
<th>x</th>
<th>df</th>
<th>nc</th>
<th>prob</th>
<th>ncc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1.0</td>
<td>0.76457</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>1.5</td>
<td>0.61893</td>
<td>1.5</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
<td>2.0</td>
<td>0.45567</td>
<td>2.0</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2.5</td>
<td>0.30115</td>
<td>2.5</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>4</td>
<td>3.0</td>
<td>0.17702</td>
<td>3.0</td>
</tr>
</tbody>
</table>
Category: Character
See: TRANSLATE Function in the documentation for your operating environment.

Syntax
TRANSLATE(source, to-1, from-1<, ...to-n, from-n>)

Arguments

source
specifies the SAS expression that contains the original character value.

to
specifies the characters that you want TRANSLATE to use as substitutes.

from
specifies the characters that you want TRANSLATE to replace.

Interaction: Values of to and from correspond on a character-by-character basis; TRANSLATE changes character one of from to character one of to, and so on. If to has fewer characters than from, TRANSLATE changes the extra from characters to blanks. If to has more characters than from, TRANSLATE ignores the extra to characters.

Operating Environment Information: You must have pairs of to and from arguments on some operating environments. On other operating environments, a segment of the collating sequence replaces null from arguments. See the SAS documentation for your operating environment for more information.

Details
If the TRANSLATE function returns a value to a variable that has not yet been assigned a length, by default the variable length is determined by the length of the first argument.

The maximum number of pairs of to and from arguments that TRANSLATE accepts depends on the operating environment you use to run SAS. There is no functional difference between using several pairs of short arguments, or fewer pairs of longer arguments.

Comparisons
The TRANWRD function differs from TRANSLATE in that it scans for words (or patterns of characters) and replaces those words with a second word (or pattern of characters).

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=translate('XYZW','AB','VW');</td>
<td>XYZB</td>
</tr>
</tbody>
</table>
See Also

Function:

“TRANWRD Function” on page 861

TRANTAB Function

Transcodes a data string by using a translation table

Category: Character


TRANWRD Function

Replaces or removes all occurrences of a word in a character string

Category: Character

Syntax

TRANWRD(source,target,replacement)

Arguments

source
specifies the source string that you want to translate.

target
specifies the string searched for in source.

replacement
specifies the string that replaces target.

Details

If the TRANWRD function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

The TRANWRD function replaces or removes all occurrences of a given word (or a pattern of characters) within a character string. The TRANWRD function does not remove trailing blanks in the target string and the replacement string.

The value that the TRANWRD function returns has a default length of 200. You can use the LENGTH statement, before calling TRANWRD, to change the length of the value.
Comparisons

The TRANSLATE function converts every occurrence of a user-supplied character to
another character. TRANSLATE can scan for more than one character in a single call.
In doing this, however, TRANSLATE searches for every occurrence of any of the
individually characters within a string. That is, if any letter (or character) in the target
string is found in the source string, it is replaced with the corresponding letter (or
character) in the replacement string.

The TRANWRD function differs from TRANSLATE in that it scans for words (or
patterns of characters) and replaces those words with a second word (or pattern of
characters).

Examples

Example 1: Replacing All Occurrences of a Word

These statements and these values produce these results:

```sas
name=tranwrd(name, "Mrs.", "Ms.");
name=tranwrd(name, "Miss", "Ms.");
put name;
```

<table>
<thead>
<tr>
<th>Values</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mrs. Joan Smith</td>
<td>Ms. Joan Smith</td>
</tr>
<tr>
<td>Miss Alice Cooper</td>
<td>Ms. Alice Cooper</td>
</tr>
</tbody>
</table>

Example 2: Removing Blanks From the Search String

In this example, the TRANWRD function does not replace the source string because the target string contains blanks.

```sas
data list;
  input salelist $;
  length target $10 replacement $3;
  target='FISH';
  replacement='NIP';
  salelist=tranwrd(salelist,target,replacement);
  put salelist;
  datalines;
  CATFISH
;
```

The LENGTH statement left-aligns TARGET and pads it with blanks to the length of
10. This causes the TRANWRD function to search for the character string ‘FISH
’ in SALELIST. Because the search fails, this line is written to the SAS log:

```
CATFISH
```

You can use the TRIM function to exclude trailing blanks from a target or
replacement variable. Use the TRIM function with TARGET:

```sas
salelist=tranwrd(salelist,trim(target),replacement);
p
```
See Also

Function:
“TRANSLATE Function” on page 859

TRIGAMMA Function

Returns the value of the Trigamma function

Category: Mathematical

Syntax
TRIGAMMA(argument)

Arguments

argument
is numeric.
Restriction: Nonpositive integers are invalid.

Details
The TRIGAMMA function returns the derivative of the DIGAMMA function. For argument > 0, the TRIGAMMA function is the second derivative of the LGAMMA function.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=trigamma(3);</td>
<td>0.3949340668</td>
</tr>
</tbody>
</table>

TRIM Function

Removes trailing blanks from character expressions and returns one blank if the expression is missing

Category: Character

Syntax
TRIM(argument)
Arguments

**argument**
specifies any SAS character expression.

Details
If the TRIM function returns a value to a variable that has not yet been assigned a length, by default the variable length is determined by the length of the first argument. TRIM copies a character argument, removes all trailing blanks, and returns the trimmed argument as a result. If the argument is blank, TRIM returns one blank. TRIM is useful for concatenating because concatenation does not remove trailing blanks.

Assigning the results of TRIM to a variable does not affect the length of the receiving variable. If the trimmed value is shorter than the length of the receiving variable, SAS pads the value with new blanks as it assigns it to the variable.

Comparisons
The TRIM and TRIMN functions are similar. TRIM returns one blank for a blank string. TRIMN returns a null string (zero blanks) for a blank string.

Examples

**Example 1: Removing Trailing Blanks**
These statements and this data line produce these results:

```sas
data test;
  input part1 $ 1-10 part2 $ 11-20;
  hasblank=part1||part2;
  noblank=trim(part1)||part2;
  put hasblank;
  put noblank;
  datalines;
```

<table>
<thead>
<tr>
<th>Data Line</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>apple sauce</td>
<td>apple sauce applesauce</td>
</tr>
</tbody>
</table>

**Example 2: Concatenating a Blank Character Expression**

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=&quot;A&quot;</td>
<td></td>
</tr>
<tr>
<td>x=&quot; &quot;; y=&quot;&quot;&gt;</td>
<td></td>
</tr>
</tbody>
</table>
TRIMN Function

Removes trailing blanks from character expressions and returns a null string (zero blanks) if the expression is missing

Category: Character

Syntax

TRIMN(argument)

Arguments

argument

specifies any SAS character expression.

Details

If the TRIMN function returns a value to a variable that has not yet been assigned a length, by default the variable length is determined by the length of the first argument.

TRIMN copies a character argument, removes all trailing blanks, and returns the trimmed argument as a result. If the argument is blank, TRIMN returns a null string. TRIMN is useful for concatenating because concatenation does not remove trailing blanks.

Assigning the results of TRIMN to a variable does not affect the length of the receiving variable. If the trimmed value is shorter than the length of the receiving variable, SAS pads the value with new blanks as it assigns it to the variable.

Comparisons

The TRIMN and TRIM functions are similar. TRIMN returns a null string (zero blanks) for a blank string. TRIM returns one blank for a blank string.
**Examples**

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>`x=&quot;A&quot;</td>
<td></td>
</tr>
<tr>
<td><code>put x;</code></td>
<td><code>AB</code></td>
</tr>
<tr>
<td><code>x=&quot; &quot;;</code></td>
<td></td>
</tr>
<tr>
<td>`z=&quot;&gt;&quot;</td>
<td></td>
</tr>
<tr>
<td><code>put z;</code></td>
<td><code>&gt;&lt;</code></td>
</tr>
</tbody>
</table>

**See Also**

Functions:
- “COMPRESS Function” on page 456
- “LEFT Function” on page 625
- “RIGHT Function” on page 788
- “TRIM Function” on page 863

---

**TRUNC Function**

Truncates a numeric value to a specified length

**Category:** Truncation

**Syntax**

TRUNC(number,length)

**Arguments**

- `number`
  - is numeric.

- `length`
  - is an integer.

**Details**

The TRUNC function truncates a full-length `number` (stored as a double) to a smaller number of bytes, as specified in `length` and pads the truncated bytes with 0s. The truncation and subsequent expansion duplicate the effect of storing numbers in less than full length and then reading them.
Comparisons
The ROUND function returns a value rounded to the nearest round-off unit. If a round-off unit is not provided, a default value of 1 is used, and the argument is rounded to the nearest integer.

Examples

```
data test;
  length x 3;
  x=1/5;
run;
data test2;
  set test;
  if x ne 1/5 then
    put 'x ne 1/5';
  if x eq trunc(1/5,3) then
    put 'x eq trunc(1/5,3)';
run;
```

The variable X is stored with a length of 3 and, therefore, each of the above comparisons is true.

UNIFORM Function

**Returns a random variate from a uniform distribution**

**Category:** Random Number

**See:** “RANUNI Function” on page 784

**Syntax**

```
UNIFORM(seed)
```

**Arguments**

- **seed**
  - is an integer.
  - **Range:** seed < 2^{31} − 1
  - **Note:** If seed ≤ 0, the time of day is used to initialize the seed stream.

UPCASE Function

**Converts all letters in an argument to uppercase**
Category: Character

Syntax

UPCASE(argument)

Arguments

argument
specifies any SAS character expression.

Details

If the UPCASE function returns a value to a variable that has not yet been assigned a length, by default the variable length is determined by the length of the first argument. The UPCASE function copies a character argument, converts all lowercase letters to uppercase letters, and returns the altered value as a result.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>name=upcase('John B. Smith');</td>
<td></td>
</tr>
<tr>
<td>put name;</td>
<td>JOHN B. SMITH</td>
</tr>
</tbody>
</table>

See Also

Functions:
“LOWCASE Function” on page 640
“PROPCASE Function” on page 737

URLDECODE Function

Returns a string that was decoded using the URL escape syntax

Category: Web Tools

Syntax

URLDECODE(argument)
Arguments

**argument**
specifies any character expression that contains a URL escape sequence, which is a three character string of the form \%nn.

Details

The URL escape syntax is used to hide characters that may otherwise be significant when used in a URL. URLDECODE also converts plus (+) characters to spaces.

Operating Environment Information: In operating environments that use EBCDIC, SAS performs an extra translation step after it recognizes an escape sequence. The specified character is assumed to be an ASCII encoding. SAS uses the transport-to-local translation table to convert this character to an EBCDIC character in operating environments that use EBCDIC. For more information see “TRANTAB= System Option” on page 1618.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1=urldecode ('abc+def'); put x1;</td>
<td>abc def</td>
</tr>
<tr>
<td>x2=urldecode ('why%3F'); put x2;</td>
<td>why?</td>
</tr>
<tr>
<td>x3=urldecode ('%41%42%43%23%31'); put x3;</td>
<td>ABC#1</td>
</tr>
</tbody>
</table>

See Also

Function:

“URLENCODE Function” on page 869

---

**URLENCODE Function**

Returns a string that was encoded using the URL escape syntax

Category: Web Tools

Syntax

URLENCODE(argument)
Arguments

argument
specifies any character expression.

Details
The URLENCODE function encodes characters that may otherwise be significant when used in a URL. This function encodes all characters except for the following:

- all alphanumeric characters
- dollar sign ($)
- dash (-)
- underscore ( _ )
- at sign (@)
- period (.)
- exclamation point (!)
- asterisk (*)
- left parenthesis ( ) and right parenthesis ( )
- comma (,).

Note: The encoded string may be longer than the original string. Ensure that you consider the additional length when you use this function.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1=urllencode ('abc def');</td>
<td>abc%20def</td>
</tr>
<tr>
<td>put x1;</td>
<td></td>
</tr>
<tr>
<td>x2=urllencode ('why?');</td>
<td>why%3F</td>
</tr>
<tr>
<td>put x2;</td>
<td></td>
</tr>
<tr>
<td>x3=urllencode ('ABC#1');</td>
<td>ABC%231</td>
</tr>
<tr>
<td>put x3;</td>
<td></td>
</tr>
</tbody>
</table>

See Also

Function:
“URLDECODE Function” on page 868

USS Function

Returns the uncorrected sum of squares
Category: Descriptive Statistics

Syntax

USS(argument-1<,argument-n>)

Arguments

argument

is numeric. At least one nonmissing argument is required. Otherwise, the function returns a missing value. If you have more than one argument, the argument list can consist of a variable list, which is preceded by OF.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1=uss(4,2,3.5,6);</td>
<td>68.25</td>
</tr>
<tr>
<td>x2=uss(4,2,3.5,6,.);</td>
<td>68.25</td>
</tr>
<tr>
<td>x3=uss(of x1-x2);</td>
<td>9316.125</td>
</tr>
</tbody>
</table>

UUIDGEN Function

Returns the short or binary form of a Universal Unique Identifier (UUID)

Category: Special

Syntax

UUIDGEN(<max-warnings<,binary-result>>) 

Arguments

max-warnings

specifies an integer value that represents the maximum number of warnings that this function writes to the log.

Default: 1

binary-result

specifies an integer value that indicates whether this function should return a binary result. Nonzero indicates a binary result should be returned. Zero indicates that a character result should be returned.

Default: 0
Details
The UUIDGEN function returns a UUID (a unique value) for each cell. The default result is 36 characters long and it looks like:

5ab6fa40--426b-4375--bb22--2d0291f43319

A binary result is 16 bytes long.

See Also
“Universal Unique Identifiers” in SAS Language Reference: Concepts

VAR Function

Returns the variance

Category: Descriptive Statistics

Syntax
VAR(argument,argument, ...)

Arguments

argument
is numeric. At least two nonmissing arguments are required. Otherwise, the function returns a missing value. The argument list can consist of a variable list, which is preceded by OF.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1=var(4,2,3.5,6);</td>
<td>2.7291666667</td>
</tr>
<tr>
<td>x2=var(4,6, .);</td>
<td>2</td>
</tr>
<tr>
<td>x3=var(of x1-x2);</td>
<td>0.2658420139</td>
</tr>
</tbody>
</table>

VARFMT Function

Returns the format assigned to a SAS data set variable

Category: SAS File I/O
Syntax

VARFMT(data-set-id, var-num)

Arguments

data-set-id
   specifies the data set identifier that the OPEN function returns.

var-num
   specifies the number of the variable’s position in the SAS data set.
   Tip: This number is next to the variable in the list that is produced by the CONTENTS procedure.
   Tip: The VARNUM function returns this number.

Details

If no format has been assigned to the variable, a blank string is returned.

Examples

Example 1: Using VARFMT to Obtain the Format of the Variable NAME
   This example obtains the format of the variable NAME in the SAS data set MYDATA.

   %let dsid=%sysfunc(open(mydata,i));
   %if &dsid %then
      %do;
      %let fmt=%sysfunc(varfmt(&dsid,
                     %sysfunc(varnum
                     (&dsid,NAME))));
      %let rc=%sysfunc(close(&dsid));
      %end;

Example 2: Using VARFMT to Obtain the Format of all the Numeric Variables in a Data Set
   This example creates a data set that contains the name and formatted content of each numeric variable in the SAS data set MYDATA.

   data vars;
      length name $ 8 content $ 12;
      drop dsid i num rc fmt;
      dsid=open("mydata","i");
      num=attrn(dsid,"nvars");
      do while (fetch(dsid)=0);
         do i=1 to num;
            name=varname(dsid,i);
            if (vartype(dsid,i)=’N’) then do;
               fmt=varfmt(dsid,i);
               if fmt=’’ then fmt="BEST12.");
               content=putc(putn(getvarn
                  (dsid,i),fmt),"$char12.");
               output;
            end;
         end;
      end;
rc=close(dsid);
run;

See Also

Functions:
   "VARINFMT Function" on page 874
   "VARNUM Function" on page 878

VARINFMT Function

Returns the informat assigned to a SAS data set variable
Category:  SAS File I/O

Syntax

VARINFMT(data-set-id, var-num)

Arguments

data-set-id
   specifies the data set identifier that the OPEN function returns.

var-num
   specifies the number of the variable’s position in the SAS data set.
   Tip:  This number is next to the variable in the list that is produced by the
         CONTENTS procedure.
   Tip:  The VARNUM function returns this number.

Details

If no informat has been assigned to the variable, a blank string is returned.

Examples

Example 1: Using VARINFMT to Obtain the Informat of the Variable NAME   This
example obtains the informat of the variable NAME in the SAS data set MYDATA.

   %let dsid=%sysfunc(open(mydata,i));
   %if &dsid %then
      %do;
      %let fmt=%sysfunc(varinfmt(&dsid,
          %sysfunc(varnum
          (&dsid,NAME))));
      %let rc=%sysfunc(close(&dsid));
      %end;
Example 2: Using VARINFMT to Obtain the Informat of all the Variables in a Data Set

This example creates a data set that contains the name and informat of the variables in MYDATA.

```sas
data vars;
  length name $ 8 informat $ 10 ;
  drop dsid i num rc;
  dsid=open("mydata","i");
  num=attrn(dsid,"nvars");
  do i=1 to num;
    name=varname(dsid,i);
    informat=varinfmt(dsid,i);
    output;
  end;
  rc=close(dsid);
run;
```

See Also

Functions:

“OPEN Function” on page 690
“VARFMT Function” on page 872
“VARNUM Function” on page 878

---

**VARLABEL Function**

Returns the label assigned to a SAS data set variable

**Category:** SAS File I/O

**Syntax**

```
VARLABEL(data-set-id,var-num)
```

**Arguments**

**data-set-id**

specifies the data set identifier that the OPEN function returns.

**var-num**

specifies the number of the variable’s position in the SAS data set.

**Tip:** This number is next to the variable in the list that is produced by the CONTENTS procedure.

**Tip:** The VARNUM function returns this number.

**Details**

If no label has been assigned to the variable, a blank string is returned.
Comparisons

VLABEL returns the label that is associated with the given variable.

Examples

This example obtains the label of the variable NAME in the SAS data set MYDATA.

Example Code 4.1  Obtaining the Label of the Variable NAME

```sas
%let dsid=%sysfunc(open(mydata,i));
%if &dsid %then
  %do;
    %let fmt=%sysfunc(varlabel(&dsid,
      %sysfunc(varnum
      (&dsid,NAME))));
    %let rc=%sysfunc(close(&dsid));
  %end;
```

See Also

Functions:

“OPEN Function” on page 690
“VARNUM Function” on page 878

VARLEN Function

Returns the length of a SAS data set variable

Category:  SAS File I/O

Syntax

**VARLEN**(data-set-id, var-num)

Arguments

*data-set-id*

specifies the data set identifier that the OPEN function returns.

*var-num*

specifies the number of the variable’s position in the SAS data set.

Tip:  This number is next to the variable in the list that is produced by the CONTENTS procedure.

Tip:  The VARNUM function returns this number.

Comparisons

VLENGTH returns the compile-time (allocated) size of the given variable.
Examples

- This example obtains the length of the variable ADDRESS in the SAS data set MYDATA.

```sas
%let dsid=%sysfunc(open(mydata,i));
%if &dsid %then
  %do;
    %let len=%sysfunc(varlen(&dsid,
                     %sysfunc(varnum
                     (&dsid,ADDRESS))));
    %let rc=%sysfunc(close(&dsid));
  %end;
```

- This example creates a data set that contains the name, type, and length of the variables in MYDATA.

```sas
data vars;
  length name $ 8 type $ 1;
  drop dsid i num rc;
  dsid=open("mydata","i");
  num=attrn(dsid,"nvars");
  do i=1 to num;
    name=varname(dsid,i);
    type=vartype(dsid,i);
    length=varlen(dsid,i);
    output;
  end;
  rc=%sysfunc(close(dsid));
run;
```

See Also

Functions:
- “OPEN Function” on page 690
- “VARNUM Function” on page 878

---

**VARNAME Function**

Returns the name of a SAS data set variable

Category: SAS File I/O

**Syntax**

```
VARNAME(data-set-id,var-num)
```
Arguments

**data-set-id**

specifies the data set identifier that the OPEN function returns.

**var-num**

specifies the number of the variable's position in the SAS data set.

**Tip:** This number is next to the variable in the list that is produced by the CONTENTS procedure.

**Tip:** The VARNUM function returns this number.

Examples

This example copies the names of the first five variables in the SAS data set CITY (or all of the variables if there are fewer than five) into a macro variable.

```sas
%let dsid=%sysfunc(open(city,i));
%let varlist=;
%do i=1 %to %sysfunc(min(5,%sysfunc(attrn(&dsid,nvars))));
  %let varlist=&varlist %sysfunc(varname(&dsid,&i));
%end;
%put varlist=&varlist;
%mend;
```

See Also

Functions:

“OPEN Function” on page 690

“VARNUM Function” on page 878

---

**VARNUM Function**

Returns the number of a variable's position in a SAS data set

**Category:** SAS File I/O

**Syntax**

```
VARNUM(data-set-id, var-name)
```

**Arguments**

**data-set-id**
specifies the data set identifier that the OPEN function returns.

var-name
specifies the variable’s name.

Details
VARNUM returns the number of a variable’s position in a SAS data set, or 0 if the variable is not in the SAS data set. This is the same variable number that is next to the variable in the output from PROC CONTENTS.

Examples

- This example obtains the number of a variable’s position in the SAS data set CITY, given the name of the variable.

```sas
%let dsid=%sysfunc(open(city,i));
%let citynum=%sysfunc(varnum(&dsid,CITYNAME));
%let rc=%sysfunc(fetch(&dsid));
%let cityname=%sysfunc(getvarc
  (&dsid,&citynum));
```

- This example creates a data set that contains the name, type, format, informat, label, length, and position of the variables in SASUSER.HOUSES.

```sas
data vars;
  length name $ 8 type $ 1
    format informat $ 10 label $ 40;
  drop dsid i num rc;
  dsid=open("sasuser.houses","i");
  num=attrn(dsid,"nvars");
  do i=1 to num;
    name=varname(dsid,i);
    type=vartype(dsid,i);
    format=varfmt(dsid,i);
    informat=varinfmt(dsid,i);
    label=varlabel(dsid,i);
    length=varlen(dsid,i);
    position=varnum(dsid,name);
    output;
  end;
  rc=close(dsid);
run;
```

See Also

Functions:
- “OPEN Function” on page 690
- “VARNUM Function” on page 877

VARRAY Function

Returns a value that indicates whether the specified name is an array
Category: Variable Information

Syntax

VARRAY (name)

Arguments

name

specifies a name that is expressed as a scalar or as an array reference.

Restriction: You cannot use an expression as an argument.

Details

VARRAY returns 1 if the given name is an array; it returns 0 if the given name is not an array.

Comparisons

- VARRAY returns a value that indicates whether the specified name is an array. VARRAYX returns a value that indicates whether the value of the specified expression is an array.
- VARRAY does not accept an expression as an argument. VARRAYX accepts expressions, but the value of the specified variable cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, informat, format, among others. For a list, see the “Variable Information” functions in “Functions and CALL Routines by Category” on page 270.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>array x(3) x1-x3;</td>
<td></td>
</tr>
<tr>
<td>a=varray(x);</td>
<td></td>
</tr>
<tr>
<td>B=varray(x1);</td>
<td></td>
</tr>
<tr>
<td>put a=;</td>
<td>a=1</td>
</tr>
<tr>
<td>put B=;</td>
<td>B=0</td>
</tr>
</tbody>
</table>

See Also

Functions:

“Variable Information” functions in “Functions and CALL Routines by Category” on page 270
VARRAYX Function

Returns a value that indicates whether the value of the specified argument is an array

Category: Variable Information

Syntax

VARRAYX (expression)

Arguments

expression
specifies any SAS character expression.

Restriction: The value of the specified expression cannot denote an array reference.

Details

VARRAYX returns 1 if the value of the given argument is the name of an array; it returns 0 if the value of the given argument is not the name of an array.

Comparisons

- VARRAY returns a value that indicates whether the specified name is the name of an array. VARRAYX returns a value that indicates whether the value of the specified expression is the name of an array.
- VARRAY does not accept an expression as an argument. VARRAYX accepts expressions, but the value of the specified variable cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, informat, format, among others. For a list, see the “Variable Information” functions in “Functions and CALL Routines by Category” on page 270.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>array x(3) x1-x3;</td>
<td></td>
</tr>
<tr>
<td>array vx(4) $6 vx1 vx2 vx3 vx4</td>
<td></td>
</tr>
<tr>
<td>('x' 'x1' 'x2' 'x3');</td>
<td></td>
</tr>
<tr>
<td>y=varrayx(vx(1));</td>
<td></td>
</tr>
<tr>
<td>z=varrayx(vx(2));</td>
<td></td>
</tr>
<tr>
<td>put y=;</td>
<td>y=1</td>
</tr>
<tr>
<td>put z=;</td>
<td>z=0</td>
</tr>
</tbody>
</table>
See Also

Functions:
“Variable Information” functions in “Functions and CALL Routines by Category” on page 270

VARTYPE Function

Returns the data type of a SAS data set variable

Category: SAS File I/O

Syntax

VARTYPE(data-set-id, var-num)

Arguments

data-set-id
specifies the data set identifier that the OPEN function returns.

var-num
specifies the number of the variable's position in the SAS data set.

Tip: This number is next to the variable in the list that is produced by the CONTENTS procedure.

Tip: The VARNUM function returns this number.

Details

VARTYPE returns C for a character variable or N for a numeric variable.

Examples

Example 1: Using VARTYPE to Determine which Variables are Numeric
This example places the names of all the numeric variables of the SAS data set MYDATA into a macro variable.
%let dsid=%sysfunc(open(mydata,i));
%let varlist=;
%do i=1 %to %sysfunc(attrn(&dsid,nvars));
  %if (%sysfunc(vartype(&dsid,&i)) = N) %then
    %let varlist=&varlist %sysfunc(varname
    (&dsid,&i));
%end;
%let rc=%sysfunc(close(&dsid));

**Example 2: Using VARTYPE to Determine which Variables are Character**  
This example creates a data set that contains the name and formatted contents of each character variable in the SAS data set MYDATA.

data vars;
  length name $ 8 content $ 20;
  drop dsid i num fmt rc;
  dsid=open("mydata","i");
  num=attrn(dsid,"nvars");
  do while (fetch(dsid)=0);
    do i=1 to num;
      name=varname(dsid,i);
      fmt=varfmt(dsid,i);
      if (vartype(dsid,i)='C') then do;
        content=getvarc(dsid,i);
        if (fmt ne '') then
          content=left(putc(content,fmt));
        output;
      end;
    end;
  end;
  rc=close(dsid);
run;

**See Also**

Function:

“VARNUM Function” on page 878

---

**VERIFY Function**

**Returns the position of the first character that is unique to an expression**

**Category:** Character

**Syntax**

```plaintext
VERIFY(source,excerpt-1<,...,excerpt-n>)
```
Arguments

*source*

specifies any SAS character expression.

*excerpt*

specifies any SAS character expression. If you specify more than one excerpt, separate them with a comma.

Details

The VERIFY function returns the position of the first character in *source* that is not present in any *excerpt*. If VERIFY finds every character in *source* in at least one *excerpt*, it returns a 0.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>data scores;</td>
<td></td>
</tr>
<tr>
<td>input Grade : $1. @@;</td>
<td></td>
</tr>
<tr>
<td>check='abcdef';</td>
<td></td>
</tr>
<tr>
<td>if verify(grade,check)&gt;0 then</td>
<td></td>
</tr>
<tr>
<td>put @1 'INVALID ' grade=;</td>
<td></td>
</tr>
<tr>
<td>datalines;</td>
<td></td>
</tr>
<tr>
<td>a b c b c d f a a q a b d d b</td>
<td>INVALID Grade=q</td>
</tr>
</tbody>
</table>

VFORMAT Function

Returns the format that is associated with the specified variable

Category: Variable Information

Syntax

VFORMAT (var)

Arguments

*var*

specifies a variable that is expressed as a scalar or as an array reference.

Restriction: You cannot use an expression as an argument.
Details

If the VFORMAT function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

VFORMAT returns the complete format name, which includes the width and the period (for example, $CHAR20.).

Comparisons

- VFORMAT returns the format that is associated with the specified variable. VFORMATX, however, evaluates the argument to determine the variable name. The function then returns the format that is associated with that variable name.
- VFORMAT does not accept an expression as an argument. VFORMATX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, type, length, among others. For a list, see the “Variable Information” functions in “Functions and CALL Routines by Category” on page 270.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>array x(3) x1-x3;</td>
<td></td>
</tr>
<tr>
<td>format x1 best6.;</td>
<td></td>
</tr>
<tr>
<td>y=vformat(x(1));</td>
<td></td>
</tr>
<tr>
<td>put y=;</td>
<td>y=BEST6.</td>
</tr>
</tbody>
</table>

See Also

Functions:

“Variable Information” functions in “Functions and CALL Routines by Category” on page 270

VFORMATD Function

Returns the format decimal value that is associated with the specified variable

Category: Variable Information

Syntax

VFORMATD (var)
Arguments

\textit{var}

specifies a variable that is expressed as a scalar or as an array reference.

\textbf{Restriction:} You cannot use an expression as an argument.

Comparisons

- VFORMATD returns the format decimal value that is associated with the specified variable. VFORMATDX, however, evaluates the argument to determine the variable name. The function then returns the format decimal value that is associated with that variable name.
- VFORMATD does not accept an expression as an argument. VFORMATDX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, type, and length, among others. For a list, see the “Variable Information” functions in “Functions and CALL Routines by Category” on page 270.

Examples

\begin{verbatim}
SAS Statements Results
array x(3) x1-x3;
format x1 comma8.2;
y=vformatd(x(1));
put y=; y=2
\end{verbatim}

See Also

Functions:

“Variable Information” functions in “Functions and CALL Routines by Category” on page 270

VFORMATDX Function

\textbf{Returns the format decimal value that is associated with the value of the specified argument}

\textbf{Category:} Variable Information

\textbf{Syntax}

\texttt{VFORMATDX (expression)}
Arguments

**expression**
specifies any SAS character expression that evaluates to a variable name.

**Restriction:** The value of the specified expression cannot denote an array reference.

Comparisons

- **VFORMATD** returns the format decimal value that is associated with the specified variable. **VFORMATDX**, however, evaluates the argument to determine the variable name. The function then returns the format decimal value that is associated with that variable name.
- **VFORMATD** does not accept an expression as an argument. **VFORMATDX** accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, length, type, among others. For a list, see the “Variable Information” functions in “Functions and CALL Routines by Category” on page 270.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>array x(3) x1-x3; format x1 comma8.2; array vx(3) $6 vx1 vx2 vx3 ('x1' 'x2' 'x3'); y=vformatdx(vx(1)); z=vformatdx('x'</td>
<td></td>
</tr>
</tbody>
</table>

See Also

Functions:

“Variable Information” functions in “Functions and CALL Routines by Category” on page 270

### VFORMATN Function

Returns the format name that is associated with the specified variable

**Category:** Variable Information

**Syntax**

VFORMATN *(var)*
Arguments

var
  specifies a variable that is expressed as a scalar or as an array reference.
  
  **Restriction:** You cannot use an expression as an argument.

Details

If the VFORMATN function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

VFORMATN returns only the format name, which does not include the width or the period (for example, $CHAR).

Comparisons

- VFORMATN returns the format name that is associated with the specified variable. VFORMATNX, however, evaluates the argument to determine the variable name. The function then returns the format name that is associated with that variable name.
- VFORMATN does not accept an expression as an argument. VFORMATNX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, type, and length, among others. For a list, see the “Variable Information” functions in “Functions and CALL Routines by Category” on page 270.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>array x(3) x1-x3;</td>
<td></td>
</tr>
<tr>
<td>format x1 best6.;</td>
<td></td>
</tr>
<tr>
<td>y=vformatn(x(1));</td>
<td></td>
</tr>
<tr>
<td>put y=;</td>
<td>y=BEST</td>
</tr>
</tbody>
</table>

See Also

Functions:
  “Variable Information” functions in “Functions and CALL Routines by Category” on page 270

VFORMATNX Function

**Returns the format name that is associated with the value of the specified argument**

**Category:** Variable Information
Syntax

VFORMATNX(expression)

Arguments

expression
specifies any SAS character expression that evaluates to a variable name.

Restriction: The value of the specified expression cannot denote an array reference.

Details

If the VFORMATNX function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200. VFORMATNX returns only the format name, which does not include the length or the period (for example, $CHAR).

Comparisons

- VFORMATN returns the format name that is associated with the specified variable. VFORMATNX, however, evaluates the argument to determine the variable name. The function then returns the format name that is associated with that variable name.
- VFORMATN does not accept an expression as an argument. VFORMATNX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, length, and type, among others. For a list, see the “Variable Information” functions in “Functions and CALL Routines by Category” on page 270.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>array x(3) x1-x3;</td>
<td></td>
</tr>
<tr>
<td>format x1 best6.;</td>
<td></td>
</tr>
<tr>
<td>array vx(3) $6 vx1 vx2 vx3</td>
<td></td>
</tr>
<tr>
<td>(’x1’ ’x2’ ’x3’);</td>
<td></td>
</tr>
<tr>
<td>y=vformatnx(vx(1));</td>
<td></td>
</tr>
<tr>
<td>put y=;</td>
<td>y=BEST</td>
</tr>
</tbody>
</table>

See Also

Functions:

“Variable Information” functions in “Functions and CALL Routines by Category” on page 270
VFORMATW Function

Returns the format width that is associated with the specified variable

Category: Variable Information

Syntax

VFORMATW (var)

Arguments

var

specifies a variable that is expressed as a scalar or as an array reference.

Restriction: You cannot use an expression as an argument.

Comparisons

- VFORMATW returns the format width that is associated with the specified variable. VFORMATWX, however, evaluates the argument to determine the variable name. The function then returns the format width that is associated with that variable name.
- VFORMATW does not accept an expression as an argument. VFORMATWX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, type, and length, among others. For a list, see the “Variable Information” functions in “Functions and CALL Routines by Category” on page 270.

Examples

SAS Statements | Results
--- | ---
array x(3) x1-x3; | 
format x1 best6.; | 
y=vformatw(x(1)); | 
put y=; | y=6 |

See Also

Functions:

“Variable Information” functions in “Functions and CALL Routines by Category” on page 270
**VFFORMATWX Function**

**Returns the format width that is associated with the value of the specified argument**

**Category:** Variable Information

**Syntax**

\[ \texttt{VFFORMATWX \langle expression \rangle} \]

**Arguments**

**expression**

specifies any SAS character expression that evaluates to a variable name.

**Restriction:** The value of the specified expression cannot denote an array reference.

**Comparisons**

- VFORMATWX returns the format width that is associated with the specified variable. VFORMATWX, however, evaluates the argument to determine the variable name. The function then returns the format width that is associated with that variable name.
- VFORMATW does not accept an expression as an argument. VFORMATWX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, length, and type, among others. For a list, see the “Variable Information” functions in “Functions and CALL Routines by Category” on page 270.

**Examples**

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>array x(3) x1-x3;</td>
<td></td>
</tr>
<tr>
<td>format x1 best6.;</td>
<td></td>
</tr>
<tr>
<td>array vx(3) $6 vx1 vx2 vx3</td>
<td></td>
</tr>
<tr>
<td>('x1' 'x2' 'x3');</td>
<td></td>
</tr>
<tr>
<td>y=VFFORMATWX(vx(1));</td>
<td>y=6</td>
</tr>
<tr>
<td>put y=;</td>
<td></td>
</tr>
</tbody>
</table>

**See Also**

Functions:

“Variable Information” functions in “Functions and CALL Routines by Category” on page 270
VFORMATX Function

Returns the format that is associated with the value of the specified argument

Category: Variable Information

Syntax

VFORMATX (expression)

Arguments

expression
specifies any SAS character expression that evaluates to a variable name.

Restriction: The value of the specified expression cannot denote an array reference.

Details

If the VFORMATX function returns a value to a variable that has not yet been assigned
a length, by default the variable is assigned a length of 200.

VFORMATX returns the complete format name which includes the width and the
period (for example, $CHAR20.).

Comparisons

- VFORMAT returns the format that is associated with the specified variable.
  VFORMATX, however, evaluates the argument to determine the variable name.
  The function then returns the format that is associated with that variable name.
- VFORMAT does not accept an expression as an argument. VFORMATX accepts
  expressions, but the value of the specified expression cannot denote an array
  reference.
- Related functions return the value of other variable attributes, such as the variable
  name, length, and type, among others. For a list, see the “Variable Information”
  functions in “Functions and CALL Routines by Category” on page 270.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>array x(3) x1-x3;</td>
<td></td>
</tr>
<tr>
<td>format x1 best6.;</td>
<td></td>
</tr>
<tr>
<td>format x2 20.10;</td>
<td></td>
</tr>
<tr>
<td>array vx(3) $6 vx1 vx2 vx3</td>
<td></td>
</tr>
<tr>
<td>{'x1' 'x2' 'x3'};</td>
<td></td>
</tr>
<tr>
<td>y=vformatx(vx(1));</td>
<td></td>
</tr>
<tr>
<td>z=vformatx(vx(2));</td>
<td></td>
</tr>
<tr>
<td>put y=; y=BEST6.</td>
<td></td>
</tr>
<tr>
<td>put z=; z=F20.10</td>
<td></td>
</tr>
</tbody>
</table>
See Also

Functions:
“Variable Information” functions in “Functions and CALL Routines by Category” on page 270

VINARRAY Function

Returns a value that indicates whether the specified variable is a member of an array

Category: Variable Information

Syntax

VINARRAY (var)

Arguments

var

specifies a variable that is expressed as a scalar or as an array reference.

Restriction: You cannot use an expression as an argument.

Details

VINARRAY returns 1 if the given variable is a member of an array; it returns 0 if the given variable is not a member of an array.

Comparisons

- VINARRAY returns a value that indicates whether the specified variable is a member of an array. VINARRAYX, however, evaluates the argument to determine the variable name. The function then returns a value that indicates whether the variable name is a member of an array.
- VINARRAY does not accept an expression as an argument. VINARRAYX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, informat, and format, among others. For a list, see the “Variable Information” functions in “Functions and CALL Routines by Category” on page 270.
**Examples**

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>array x(3) x1-x3; y=vinarray(x); Z=vinarray(x1); put y=; put Z=;</td>
<td>y=0  z=1</td>
</tr>
</tbody>
</table>

**See Also**

Functions:

“Variable Information” functions in “Functions and CALL Routines by Category” on page 270

---

**VINARRAYX Function**

Returns a value that indicates whether the value of the specified argument is a member of an array

**Category:** Variable Information

**Syntax**

VINARRAYX *(expression)*

**Arguments**

*expression*

specifies any SAS character expression that evaluates to a variable name.

**Restriction:** The value of the specified expression cannot denote an array reference.

**Details**

VINARRAYX returns 1 if the value of the given argument is a member of an array; it returns 0 if the value of the given argument is not a member of an array.

**Comparisons**

VINARRAYX returns a value that indicates whether the specified variable is a member of an array. VINARRAYX, however, evaluates the argument to determine the variable name. The function then returns a value that indicates whether the variable name is a member of an array.
VINARRAY does not accept an expression as an argument. VINARRAYX accepts expressions, but the value of the specified expression cannot denote an array reference.

Related functions return the value of other variable attributes, such as the variable name, informat, and format, among others. For a list, see the “Variable Information” functions in “Functions and CALL Routines by Category” on page 270.

Examples

SAS Statements Results
array x(3) x1-x3;
array vx(4) $6 vx1 vx2 vx3 vx4 ('x' 'x1' 'x2' 'x3');
y=vinarrayx(vx(1));
z=vinarrayx(vx(2));
put y=;
put z=;
y=0
z=1

See Also

Functions:
“Variable Information” functions in “Functions and CALL Routines by Category” on page 270

VINFORMAT Function

Returns the informat that is associated with the specified variable

Category: Variable Information

Syntax

VINFORMAT (var)

Arguments

var
specifies a variable that is expressed as a scalar or as an array reference.

Restriction: You cannot use an expression as an argument.

Details

If the VINFORMAT function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.
VINFORMAT returns the complete informat name, which includes the width and the period (for example, $CHAR20.).

**Comparisons**

- VINFORMAT returns the informat that is associated with the specified variable. VINFORMATX, however, evaluates the argument to determine the variable name. The function then returns the informat that is associated with that variable name.
- VINFORMAT does not accept an expression as an argument. VINFORMATX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, type, and length, among others. For a list, see the “Variable Information” functions in “Functions and CALL Routines by Category” on page 270.

**Examples**

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>informat x $char6.;</td>
<td></td>
</tr>
<tr>
<td>input x;</td>
<td></td>
</tr>
<tr>
<td>y=vinformat(x);</td>
<td></td>
</tr>
<tr>
<td>put y=;</td>
<td>y=$CHAR6.</td>
</tr>
</tbody>
</table>

**See Also**

Functions:

“Variable Information” functions in “Functions and CALL Routines by Category” on page 270

---

**VINFORMATD Function**

Returns the informat decimal value that is associated with the specified variable

**Category:** Variable Information

**Syntax**

VINFORMATD (var)

**Arguments**

var

specifies a variable that is expressed as a scalar or as an array reference.
Restriction: You cannot use an expression as an argument.

Comparisons

- VINFORMATD returns the informat decimal value that is associated with the specified variable. VINFORMATDX, however, evaluates the argument to determine the variable name. The function then returns the informat decimal value that is associated with that variable name.
- VINFORMATD does not accept an expression as an argument. VINFORMATDX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, type, and length, among others. For a list, see the “Variable Information” functions in “Functions and CALL Routines by Category” on page 270.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>informat x comma8.2;</code></td>
<td></td>
</tr>
<tr>
<td><code>input x;</code></td>
<td></td>
</tr>
<tr>
<td><code>y=vinformatd(x);</code></td>
<td></td>
</tr>
<tr>
<td><code>put y=;</code></td>
<td><code>y=2</code></td>
</tr>
</tbody>
</table>

See Also

Functions:

“Variable Information” functions in “Functions and CALL Routines by Category” on page 270

VINFORMATDX Function

Returns the informat decimal value that is associated with the value of the specified argument

Category: Variable Information

Syntax

`VINFORMATDX (expression)`

Arguments

`expression` specifies any SAS character expression that evaluates to a variable name.
**Restriction:** The value of the specified variable cannot denote an array reference.

**Comparisons**

- VINFORMATD returns the informat decimal value that is associated with the specified variable. VINFORMATDX, however, evaluates the argument to determine the variable name. The function then returns the informat decimal value that is associated with that variable name.
- VINFORMATD does not accept an expression as an argument. VINFORMATDX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, length, and type, among others. For a list, see the “Variable Information” functions in “Functions and CALL Routines by Category” on page 270.

**Examples**

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>informat x1 x2 x3 comma9.3; input x1 x2 x3; array vx(3) $6 vx1 vx2 vx3 ('x1' 'x2' 'x3'); y=vinformatdx(vx(1)); put y=;</td>
<td>y=3</td>
</tr>
</tbody>
</table>

**See Also**

Functions:

“Variable Information” functions in “Functions and CALL Routines by Category” on page 270

---

**VINFORMATN Function**

Returns the informat name that is associated with the specified variable

**Category:** Variable Information

**Syntax**

VINFORMATN (var)

**Arguments**

var
specifies a variable that is expressed as a scalar or as an array reference.

**Restriction:** You cannot use an expression as an argument.

### Details

If the VINFORMATN function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

VINFORMATN returns only the informat name, which does not include the width or the period (for example, $CHAR).

### Comparisons

- VINFORMATN returns the informat name that is associated with the specified variable. VINFORMATNX, however, evaluates the argument to determine the variable name. The function then returns the informat name that is associated with that variable name.
- VINFORMATN does not accept an expression as an argument. VINFORMATNX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, type, and length, among others. For a list, see the “Variable Information” functions in “Functions and CALL Routines by Category” on page 270.

### Examples

**SAS Statements**

```sas
informat x $char6.;
input x;
y=vinformatn(x);
put y=; y=$CHAR
```

**Results**

```
y=$CHAR
```

### See Also

Functions:

“Variable Information” functions in “Functions and CALL Routines by Category” on page 270

---

**VINFORMATNX Function**

Returns the informat name that is associated with the value of the specified argument

**Category:** Variable Information

**Syntax**

VINFORMATNX *(expression)*
Arguments

*expression*

specifies any SAS character expression that evaluates to a variable name.

**Restriction:** The value of the specified expression cannot denote an array reference.

Details

If the VINFORMATNX function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

VINFORMATNX returns only the informat name, which does not include the width or the period (for example, $CHAR).

Comparisons

- VINFORMATN returns the informat name that is associated with the specified variable. VINFORMATNX, however, evaluates the argument to determine the variable name. The function then returns the informat name that is associated with that variable name.
- VINFORMATN does not accept an expression as an argument. VINFORMATNX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, length, and type, among others. For a list, see the “Variable Information” functions in “Functions and CALL Routines by Category” on page 270.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>informat x1 x2 x3 $char6.;</code></td>
<td><code>y=$CHAR</code></td>
</tr>
<tr>
<td><code>input x1 x2 x3;</code></td>
<td></td>
</tr>
<tr>
<td><code>array vx(3) $6 vx1 vx2 vx3</code></td>
<td></td>
</tr>
<tr>
<td><code>('x1' 'x2' 'x3');</code></td>
<td></td>
</tr>
<tr>
<td><code>y=vinformatnx(vx(1));</code></td>
<td></td>
</tr>
<tr>
<td><code>put y=;</code></td>
<td></td>
</tr>
</tbody>
</table>

See Also

Functions:

“Variable Information” functions in “Functions and CALL Routines by Category” on page 270

VINFORMATW Function

Returns the informat width that is associated with the specified variable
VINFORMATWX Function

Returns the informat width that is associated with the value of the specified argument.
Category: Variable Information

Syntax
VINFORMATWX (expression)

Arguments

expression specifies any SAS character expression that evaluates to a variable name.
Restriction: The value of the specified expression cannot denote an array reference.

Comparisons

- VINFORMATW returns the informat width that is associated with the specified variable. VINFORMATWX, however, evaluates the argument to determine the variable name. The function then returns the informat width that is associated with that variable name.
- VINFORMATW does not accept an expression as an argument. VINFORMATWX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, length, and type, among others. For a list, see the “Variable Information” functions in “Functions and CALL Routines by Category” on page 270.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>informat x1 x2 x3 $char6.;</td>
<td></td>
</tr>
<tr>
<td>input x1 x2 x3;</td>
<td></td>
</tr>
<tr>
<td>array vx(3) $6 vx1 vx2 vx3 ('x1' 'x2' 'x3');</td>
<td></td>
</tr>
<tr>
<td>y=vinformatwx(vx(1));</td>
<td></td>
</tr>
<tr>
<td>put y=;</td>
<td>y=6</td>
</tr>
</tbody>
</table>

See Also

Functions:
“Variable Information” functions in “Functions and CALL Routines by Category” on page 270

VINFORMATX Function

Returns the informat that is associated with the value of the specified argument
Category: Variable Information

Syntax

VINFORMATX (expression)

Arguments

expression
specifies any SAS character expression that evaluates to a variable name.
Restriction: The value of the specified expression cannot denote an array reference.

Details

If the VINFORMATX function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200. VINFORMATX returns the complete informat name, which includes the width and the period (for example, $CHAR20.).

Comparisons

- VINFORMAT returns the informat that is associated with the specified variable. VINFORMATX, however, evaluates the argument to determine the variable name. The function then returns the informat that is associated with that variable name.
- VINFORMAT does not accept an expression as an argument. VINFORMATX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, length, and type, among others. For a list, see the “Variable Information” functions in “Functions and CALL Routines by Category” on page 270.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>informat x1 x2 x3 $char6.;</td>
<td></td>
</tr>
<tr>
<td>input x1 x2 x3;</td>
<td></td>
</tr>
<tr>
<td>array vx(3) $6 vx1 vx2 vx3</td>
<td></td>
</tr>
<tr>
<td>('x1' 'x2' 'x3');</td>
<td></td>
</tr>
<tr>
<td>y=vinformatx(vx(1));</td>
<td>y=$CHAR6.</td>
</tr>
<tr>
<td>put y=;</td>
<td></td>
</tr>
</tbody>
</table>

See Also

Functions:

“Variable Information” functions in “Functions and CALL Routines by Category” on page 270
VLABEL Function

Returns the label that is associated with the specified variable

Category: Variable Information

Syntax

VLABEL (var)

Arguments

var

specifies a variable that is expressed as a scalar or as an array reference.

Restriction: You cannot use an expression as an argument.

Details

If the VLABEL function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

If there is no label, VLABEL returns the variable name.

Comparisons

- VLABEL returns the label of the specified variable or the name of the specified variable, if no label exists. VLABELX, however, evaluates the argument to determine the variable name. The function then returns the label that is associated with that variable name, or the variable name if no label exists.
- VLABEL does not accept an expression as an argument. VLABELX accepts expressions, but the value of the specified expression cannot denote an array reference.
- VLABEL has the same functionality as CALL LABEL.
- Related functions return the value of other variable attributes, such as the variable name, informat, and format, among others. For a list, see the “Variable Information” functions in “Functions and CALL Routines by Category” on page 270.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>array x(3) x1-x3; label x1='Test1'; y=vlabel(x(1)); put y=;</td>
<td>y=Test1</td>
</tr>
</tbody>
</table>
See Also

Functions:
“Variable Information” functions in “Functions and CALL Routines by Category”
on page 270

VLABELX Function

Returns the variable label for the value of the specified argument

Category: Variable Information

Syntax
VLABELX (expression)

Arguments

expression specifies any SAS character expression that evaluates to a variable name.

Restriction: The value of the specified expression cannot denote an array reference.

Details

If the VLABELX function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

If there is no label, VLABELX returns the variable name.

Comparisons

- VLABEL returns the label of the specified variable, or the name of the specified variable if no label exists. VLABELX, however, evaluates the argument to determine the variable name. The function then returns the label that is associated with that variable name, or the variable name if no label exists.
- VLABEL does not accept an expression as an argument. VLABELX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, informat, and format, among others. For a list, see the “Variable Information” functions in “Functions and CALL Routines by Category” on page 270.
Examples

SAS Statements Results

array x(3) x1-x3;
array vx(3) $6 vx1 vx2 vx3
   ('x1' 'x2' 'x3');
label x1='Test1';
y=vlabelx(vx(1));
put y=; y=Test1

See Also

Functions:
“Variable Information” functions in “Functions and CALL Routines by Category” on page 270

VLENGTH Function

Returns the compile-time (allocated) size of the specified variable

Category: Variable Information

Syntax

VLENGTH (var)

Arguments

var
specifies a variable that is expressed as a scalar or as an array reference.

Restriction: You cannot use an expression as an argument.

Comparisons

- LENGTH examines the variable at run-time, trimming trailing blanks to determine the length. VLENGTH returns a compile-time constant value, which reflects the maximum length.
- VLENGTH returns the length of the specified variable. VLENGTHX, however, evaluates the argument to determine the variable name. The function then returns the compile-time size that is associated with that variable name.
- VLENGTH does not accept an expression as an argument. VLENGTHX accepts expressions, but the value of the specified expression cannot denote an array reference.
Related functions return the value of other variable attributes, such as the variable name, informat, and format, among others. For a list, see the “Variable Information” functions in “Functions and CALL Routines by Category” on page 270.

### Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>length x $8;</td>
<td></td>
</tr>
<tr>
<td>x='abc';</td>
<td></td>
</tr>
<tr>
<td>y=vlength(x);</td>
<td>y=8</td>
</tr>
<tr>
<td>z=length(x);</td>
<td>z=3</td>
</tr>
<tr>
<td>put y=;</td>
<td></td>
</tr>
<tr>
<td>put z=;</td>
<td></td>
</tr>
</tbody>
</table>

### See Also

Functions:

“Variable Information” functions in “Functions and CALL Routines by Category” on page 270

---

**VLENGTHX Function**

Returns the compile-time (allocated) size for the value of the specified argument

**Category:** Variable Information

### Syntax

VLENGTHX(expression)

### Arguments

**expression**

specifies any SAS character expression that evaluates to a variable name.

**Restriction:** The value of the specified expression cannot denote an array reference.

### Comparisons

- LENGTH examines the variable at run-time, trimming trailing blanks to determine the length. VLENGTHX, however, evaluates the argument to determine the variable name. The function then returns the compile-time size that is associated with that variable name.
VLENGTH returns the length of the specified variable. VLENGTHX returns the length for the value of the specified expression.

VLENGTH does not accept an expression as an argument. VLENGTHX accepts expressions, but the value of the specified expression cannot denote an array reference.

Related functions return the value of other variable attributes, such as the variable name, informat, format, among others. For a list, see the “Variable Information” functions in “Functions and CALL Routines by Category” on page 270.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>length x1 $8;</td>
<td></td>
</tr>
<tr>
<td>x1='abc';</td>
<td></td>
</tr>
<tr>
<td>array vx(3) $6 vx1 vx2 vx3</td>
<td></td>
</tr>
<tr>
<td>('x1' 'x2' 'x3');</td>
<td></td>
</tr>
<tr>
<td>y=vlengthx(vx(1));</td>
<td></td>
</tr>
<tr>
<td>z=length(x1);</td>
<td></td>
</tr>
<tr>
<td>put y=;</td>
<td>y=8</td>
</tr>
<tr>
<td>put z=;</td>
<td>z=3</td>
</tr>
</tbody>
</table>

See Also

Functions:

“Variable Information” functions in “Functions and CALL Routines by Category” on page 270

VNAME Function

Returns the name of the specified variable

Category: Variable Information

Syntax

VNAME (var)

Arguments

var

specifies a variable that is expressed as a scalar or as an array reference.

Restriction: You cannot use an expression as an argument.
Details

If the VNAME function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

Comparisons

- VNAME returns the name of the specified variable. VNAMEX, however, evaluates the argument to determine a variable name. If the name is a known variable name, the function returns that name. Otherwise, the function returns a blank.
- VNAME does not accept an expression as an argument. VNAMEX accepts expressions, but the value of the specified expression cannot denote an array reference.
- VNAME has the same functionality as CALL VNAME.
- Related functions return the value of other variable attributes, such as the variable label, informat, and format, among others. For a list, see the “Variable Information” functions in “Functions and CALL Routines by Category” on page 270.

Examples

SAS Statements | Results
---|---
array x(3) x1-x3;
y=vname(x(1));
put y=; | y=x1

See Also

Functions:

“Variable Information” functions in “Functions and CALL Routines by Category” on page 270

VNAMEX Function

Validates the value of the specified argument as a variable name

Category: Variable Information

Syntax

VNAMEX (expression)
Arguments

expression

specifies any SAS character expression.

Restriction: The value of the specified expression cannot denote an array reference.

Details

If the VNAMEX function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

Comparisons

- VNAME returns the name of the specified variable. VNAMEX, however, evaluates the argument to determine a variable name. If the name is a known variable name, the function returns that name. Otherwise, the function returns a blank.
- VNAME does not accept an expression as an argument. VNAMEX accepts expressions, but the value of the specified variable cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable label, informat, and format, among others. For a list, see the “Variable Information” functions in “Functions and CALL Routines by Category” on page 270.

Examples

SAS Statements Results

array x(3) x1-x3;
array vx(3) $6 vx1 vx2 vx3
    ('x1' 'x2' 'x3');
y=vnamex(vx(1));
z=vnamex('x'||'1');
put y=;    y=x1
put z=;    z=x1

See Also

Functions:

“Variable Information” functions in “Functions and CALL Routines by Category” on page 270

VTRANSCODE Function

Returns a value that indicates whether transcoding is on or off for the specified character variable
Functions and CALL Routines  △  VTYPE Function

Category: Variable Information

VTRANSCODEX Function

Returns a value that indicates whether transcoding is on or off for the specified argument

Category: Variable Information

VTYPE Function

Returns the type (character or numeric) of the specified variable

Category: Variable Information

Syntax

VTYPE (var)

Arguments

var
specifies a variable that is expressed as a scalar or as an array reference.

Restriction: You cannot use an expression as an argument.

Details

If the VTYPE function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 1.

VTYPE returns N for numeric variables and C for character variables.

Comparisons

□ VTYPE returns the type of the specified variable. VTYPEX, however, evaluates the argument to determine the variable name. The function then returns the type (character or numeric) that is associated with that variable name.
□ VTYPE does not accept an expression as an argument. VTYPEX accepts expressions, but the value of the specified expression cannot denote an array reference.
□ Related functions return the value of other variable attributes, such as the variable name, informat, and format, among others. For a list, see the “Variable
Information” functions in “Functions and CALL Routines by Category” on page 270.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>array x(3) x1-x3; y=vtype(x(1)); put y=;</td>
<td>y=N</td>
</tr>
</tbody>
</table>

See Also

Functions:
“Variable Information” functions in “Functions and CALL Routines by Category” on page 270

VTYPEX Function

Returns the type (character or numeric) for the value of the specified argument

Category: Variable Information

Syntax

VTYPEX (expression)

Arguments

expression
specifies any SAS character expression that evaluates to a variable name.

Restriction: The value of the specified expression cannot denote an array reference.

Details

If the VTYPEX function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 1.

VTYPEX returns N for numeric variables and C for character variables.

Comparisons

- VTYPE returns the type of the specified variable. VTYPEX, however, evaluates the argument to determine the variable name. The function then returns the type (character or numeric) that is associated with that variable name.
VTYPE does not accept an expression as an argument. VTYPEX accepts expressions, but the value of the specified expression cannot denote an array reference.

Related functions return the value of other variable attributes, such as the variable name, informat, and format, among others. For a list, see the “Variable Information” functions in “Functions and CALL Routines by Category” on page 270.

Examples

SAS Statements Results

```sas
array x(3) x1-x3;
array vx(3) $6 vx1 vx2 vx3
('x1' 'x2' 'x3');
y=vtypex(vx(1));
put y=;      y=N
```

See Also

Functions:

“Variable Information” functions in “Functions and CALL Routines by Category” on page 270

VVALUE Function

Returns the formatted value that is associated with the variable that you specify

Category: Variable Information

Syntax

```
VVALUE(var)
```

Arguments

`var`

specifies a variable that is expressed as a scalar or as an array reference.

Restriction: You cannot use an expression as an argument.

Details

If the VVALUE function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.
VVALUE returns a character string that contains the current value of the variable that you specify. The value is formatted using the current format that is associated with the variable.

**Comparisons**

- VVALUE returns the value that is associated with the variable that you specify. VVALUEX, however, evaluates the argument to determine the variable name. The function then returns the value that is associated with that variable name.
- VVALUE does not accept an expression as an argument. VVALUEX accepts expressions, but the value of the expression cannot denote an array reference.
- VVALUE and an assignment statement both return a character string that contains the current value of the variable that you specify. With VVALUE, the value is formatted using the current format that is associated with the variable. With an assignment statement, however, the value is unformatted.
- The PUT function allows you to reformat a specified variable or constant. VVALUE uses the current format that is associated with the variable.

**Examples**

```
SAS Statements                Results
y=9999;
format y comma10.2;
v=vvalue(y);
put v;  9,999.00
```

**See Also**

Functions:

- “VVALUEX Function” on page 914
- “Variable Information” functions in “Functions and CALL Routines by Category” on page 270

---

**VVALUEX Function**

Returns the formatted value that is associated with the argument that you specify

**Category:** Variable Information

**Syntax**

`VVALUEX(expression)`
Arguments

expression
specifies any SAS character expression that evaluates to a variable name.

Restriction: The value of the specified expression cannot denote an array reference.

Details

If the VVALUEX function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

VVALUEX returns a character string that contains the current value of the argument that you specify. The value is formatted by using the format that is currently associated with the argument.

Comparisons

- VVALUE accepts a variable as an argument and returns the value of that variable. VVALUEX, however, accepts a character expression as an argument. The function then evaluates the expression to determine the variable name and returns the value that is associated with that variable name.
- VVALUE does not accept an expression as an argument, but it does accept array references. VVALUEX accepts expressions, but the value of the expression cannot denote an array reference.
- VVALUEX and an assignment statement both return a character string that contains the current value of the variable that you specify. With VVALUEX, the value is formatted by using the current format that is associated with the variable. With an assignment statement, however, the value is unformatted.
- The PUT function allows you to reformat a specified variable or constant. VVALUEX uses the current format that is associated with the variable.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>date1='31mar02'd;</td>
<td></td>
</tr>
<tr>
<td>date2='date1';</td>
<td></td>
</tr>
<tr>
<td>format date1 date7.;</td>
<td></td>
</tr>
<tr>
<td>datevalue=vvaluex(date2);</td>
<td></td>
</tr>
<tr>
<td>put datevalue;</td>
<td>31MAR02</td>
</tr>
</tbody>
</table>

See Also

Functions:

“VVALUE Function” on page 913

“Variable Information” functions in “Functions and CALL Routines by Category” on page 270
WEEK Function

Returns the week number value
Category: Date and Time

WEEKDAY Function

Returns the day of the week from a SAS date value
Category: Date and Time

Syntax
WEEKDAY(date)

Arguments

date
  specifies a SAS expression that represents a SAS date value.

Details
The WEEKDAY function produces an integer that represents the day of the week, where 1=Sunday, 2=Monday, ..., 7=Saturday.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=weekday(‘16mar97’d);</td>
<td></td>
</tr>
<tr>
<td>put x;</td>
<td>1</td>
</tr>
</tbody>
</table>

YEAR Function

Returns the year from a SAS date value
Category: Date and Time
Syntax

YEAR(date)

Arguments

date specifies a SAS expression that represents a SAS date value.

Details

The YEAR function produces a four-digit numeric value that represents the year.

Examples

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>date='25dec97'd; y=year(date); put y;</td>
<td>1997</td>
</tr>
</tbody>
</table>

See Also

Functions:

“DAY Function” on page 481
“MONTH Function” on page 655

YIELDP Function

Returns the yield-to-maturity for a periodic cash flow stream, such as a bond

Category: Financial

Syntax

YIELDP(A,c,n,K,k,u,p)

Arguments

A specifies the face value.

Range: A > 0
\( c \)
specifies the nominal annual coupon rate, expressed as a fraction.
**Range:** \( 0 \leq c < 1 \)

\( n \)
specifies the number of coupons per year.
**Range:** \( n > 0 \) and is an integer

\( K \)
specifies the number of remaining coupons from settlement date to maturity.
**Range:** \( K > 0 \) and is an integer

\( k_0 \)
specifies the time from settlement date to the next coupon as a fraction of the annual basis.
**Range:** \( 0 < k_0 \leq \frac{1}{n} \)

\( p \)
specifies the price with accrued interest.
**Range:** \( p > 0 \)

**Details**
The YIELDP function is based on the relationship

\[
P = \sum_{k=1}^{K} c(k) \frac{1}{(1 + \frac{c}{n})^{t_k}}
\]

where
\[
\begin{align*}
t_k &= nk_0 + k - 1 \\
c(k) &= \frac{c}{n} A \quad \text{for } k = 1, \ldots, K - 1 \\
c(K) &= (1 + \frac{c}{n}) A
\end{align*}
\]

The YIELDP function solves for \( y \).

**Examples**
The following example demonstrates the use of YIELDP with a face value of 1000, an annual coupon rate of 0.01, 4 coupons per year, 14 remaining coupons, time from settlement date to next coupon is 0.165, and the price with accrued interest is 800.

```r
data _null_; 
  y=yieldp(1000,.01,4,14,.165,800); 
  put y; 
run;
```
The value returned is 0.0775.

---

**YRDIF Function**

Returns the difference in years between two dates.
Category: Date and Time

Syntax

\texttt{YRDIF(sdate, edate, basis)}

Arguments

\texttt{sdate}

specifies a SAS date value that identifies the starting date.

\texttt{edate}

specifies a SAS date value that identifies the ending date.

\texttt{basis}

identifies a character constant or variable that describes how SAS calculates the date difference. The following character strings are valid:

'30/360'

specifies a 30-day month and a 360-day year in calculating the number of years. Each month is considered to have 30 days, and each year 360 days, regardless of the actual number of days in each month or year.

\textbf{Alias:} '360'

\textbf{Tip:} If either date falls at the end of a month, it is treated as if it were the last day of a 30-day month.

'ACT/ACT'

uses the actual number of days between dates in calculating the number of years. SAS calculates this value as the number of days that fall in 365-day years divided by 365 plus the number of days that fall in 366-day years divided by 366.

\textbf{Alias:} 'Actual'

'ACT/360'

uses the actual number of days between dates in calculating the number of years. SAS calculates this value as the number of days divided by 360, regardless of the actual number of days in each year.

'ACT/365'

uses the actual number of days between dates in calculating the number of years. SAS calculates this value as the number of days divided by 365, regardless of the actual number of days in each year.

Examples

In the following example, YRDIF returns the difference in years between two dates based on each of the options for \textit{basis}.

\begin{verbatim}
   data _null_
   sdate='16oct1998'd;
   edate='16feb2003'd;
   y30360=yrdif(sdate, edate, '30/360');
   yactact=yrdif(sdate, edate, 'ACT/ACT');
   yact360=yrdif(sdate, edate, 'ACT/360');
   yact365=yrdif(sdate, edate, 'ACT/365');
   put y30360= yactact= yact360= yact365=;
\end{verbatim}
YYQ Function

Returns a SAS date value from the year and quarter

Category:  Date and Time

Syntax

YYQ(year,quarter)

Arguments

year
  specifies a two-digit or four-digit integer that represents the year. The
  YEARCUTOFF= system option defines the year value for two-digit dates.

quarter
  specifies the quarter of the year (1, 2, 3, or 4).

Details

The YYQ function returns a SAS date value that corresponds to the first day of the
specified quarter. If either year or quarter is missing, or if the quarter value is not
valid, the result is missing.

SAS Statements           Results
---                    -------
put y30360=;           4.333333333
put yactact=;           4.3369863014
put yact360=;           4.4
put yact365=;           4.3397260274

See Also

Functions:
  “DATDIF Function” on page 477
Examples

SAS Statements  Results
---
DateValue=yyq(2001,3);
put DateValue;
put DateValue date7.;  01JUL01
put DateValue date9.;  01JUL2001
StartOfQtr=yyq(99,4);
put StartOfQtr;
put StartOfQtr=worddate.;  StartOfQtr=October 1, 1999

See Also

Functions:
“QTR Function” on page 760
“YEAR Function” on page 916
System Option:
“YEARCUTOFF= System Option” on page 1630

ZIPCITY Function

Returns a city name and the two-character postal code that corresponds to a ZIP code
Category:  State and ZIP Code

Syntax
ZIPCITY(zip-code)

Arguments

zip-code
specifies a numeric or character expression that contains a five-digit ZIP code.
Tip:  If the value of zip-code begins with leading zeros, you can enter the value without the leading zeroes. For example, if you enter 1040, ZIPCITY assumes that the value is 01040.

Details
If the ZIPCITY function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 20.
ZIPCITY returns a city name and the two-character postal code that corresponds to its five-digit ZIP code argument. ZIPCITY returns the character values in uppercase. If the ZIP code is unknown, ZIPCITY returns a blank value.

*Note:* The SASHELP.ZIPCODE data set must be present when you use this function. If you remove the data set, ZIPCITY will return unexpected results.

**Comparisons**

The ZIPCITY, ZIPNAME, ZIPNAMEL, and ZIPSTATE functions take the same argument but return different values:

- ZIPCITY returns the uppercase name of the city and the two-character postal code that corresponds to its five-digit ZIP code argument.
- ZIPNAME returns the uppercase name of the state or U.S. territory that corresponds to its five-digit ZIP code argument.
- ZIPNAMEL returns the mixed case name of the state or U.S. territory that corresponds to its five-digit ZIP code argument.
- ZIPSTATE returns the uppercase two-character state postal code (or world-wide GSA geographic code for U.S. territories) that corresponds to its five-digit ZIP code argument.

**Examples**

The following SAS statements produce these results.

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=zipcity(27511); put a;</td>
<td>CARY, NC</td>
</tr>
<tr>
<td>b=zipcity(58501); put b;</td>
<td>BISMARCK, ND</td>
</tr>
<tr>
<td>c=zipcity(4338); put c;</td>
<td>AUGUSTA, ME</td>
</tr>
<tr>
<td>d=zipcity(01040); put d;</td>
<td>HOLYOKE, MA</td>
</tr>
</tbody>
</table>

**See Also**

Functions:

- “ZIPNAME Function” on page 924
- “ZIPNAMEL Function” on page 925
- “ZIPSTATE Function” on page 926
- “ZIPFIPS Function” on page 922

**ZIPFIPS Function**

Converts ZIP codes to two-digit FIPS codes
Functions and CALL Routines

ZIPFIPS Function

Category: State and ZIP Code

Syntax

ZIPFIPS(zip-code)

Arguments

zip-code

specifies a numeric or character expression that contains a five-digit ZIP code.

Tip: If the value of zip-code begins with leading zeros, you can enter the value without the leading zeroes. For example, if you enter 1040, ZIPFIPS assumes that the value is 01040.

Details

The ZIPFIPS function returns the two-digit numeric U.S. Federal Information Processing Standards (FIPS) code that corresponds to its five-digit ZIP code argument.

Examples

The following SAS statements produce these results.

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>fips1=zipfips('27511'); put fips1;</td>
<td>37</td>
</tr>
<tr>
<td>fips2=zipfips('01040'); put fips2;</td>
<td>25</td>
</tr>
<tr>
<td>fips3=zipfips(1040); put fips3;</td>
<td>25</td>
</tr>
<tr>
<td>fips4=zipfips(59017); put fips4;</td>
<td>30</td>
</tr>
<tr>
<td>fips5=zipfips(24862); put fips5;</td>
<td>54</td>
</tr>
</tbody>
</table>

See Also

Functions:

“ZIPCITY Function” on page 921
“ZIPNAME Function” on page 924
“ZIPNAMEL Function” on page 925
“ZIPSTATE Function” on page 926
ZIPNAME Function

Converts ZIP codes to uppercase state names

Category: State and ZIP Code

Syntax

ZIPNAME(zip-code)

Arguments

zip-code

specifies a numeric or character expression that contains a five-digit ZIP code.

Tip: If the value of zip-code begins with leading zeros, you can enter the value without the leading zeroes. For example, if you enter 1040, ZIPNAME assumes that the value is 01040.

Details

If the ZIPNAME function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 20. ZIPNAME returns the name of the state or U.S. territory that corresponds to its five-digit ZIP code argument. ZIPNAME returns character values up to 20 characters long, all in uppercase.

Comparisons

The ZIPCITY, ZIPNAME, ZIPNAMEL, and ZIPSTATE functions take the same argument but return different values:

- ZIPCITY returns the uppercase name of the city and the two-character postal code that corresponds to its five-digit ZIP code argument.
- ZIPNAME returns the uppercase name of the state or U.S. territory that corresponds to its five-digit ZIP code argument.
- ZIPNAMEL returns the mixed case name of the state or U.S. territory that corresponds to its five-digit ZIP code argument.
- ZIPSTATE returns the uppercase two-character state postal code (or world-wide GSA geographic code for U.S. territories) that corresponds to its five-digit ZIP code argument.

Examples

The following SAS statements produce these results.
SAS Statements | Results
--- | ---
state1=zipcode('27511'); put state1; | NORTH CAROLINA
state2=zipcode('01040'); put state2; | MASSACHUSETTS
state3=zipcode(1040); put state3; | MASSACHUSETTS
state4=zipcode('59017'); put state4; | MONTANA
state5=zipcode(24862); put state5; | WEST VIRGINIA

See Also

Functions:
“ZIPCITY Function” on page 921
“ZIPFIPS Function” on page 922
“ZIPNAMEL Function” on page 925
“ZIPSTATE Function” on page 926

ZIPNAMEL Function

Converts ZIP codes to mixed case state names
Category: State and ZIP Code

Syntax

ZIPNAMEL(zip-code)

Arguments

zip-code
specifies a numeric or character expression that contains a five-digit ZIP code.
Tip: If the value of zip-code begins with leading zeros, you can enter the value without the leading zeroes. For example, if you enter 1040, ZIPNAMEL assumes that the value is 01040.

Details

If the ZIPNAMEL function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 20.
ZIPNAMEL returns the name of the state or U.S. territory that corresponds to its five-digit ZIP code argument. ZIPNAMEL returns mixed case character values up to 20 characters long.

**Comparisons**

The ZIPCITY, ZIPNAME, ZIPNAMEL, and ZIPSTATE functions take the same argument but return different values:

- ZIPCITY returns the uppercase name of the city and the two-character postal code that corresponds to its five-digit ZIP code argument.
- ZIPNAME returns the uppercase name of the state or U.S. territory that corresponds to its five-digit ZIP code argument.
- ZIPNAMEL returns the mixed case name of the state or U.S. territory that corresponds to its five-digit ZIP code argument.
- ZIPSTATE returns the uppercase two-character state postal code (or world-wide GSA geographic code for U.S. territories) that corresponds to its five-digit ZIP code argument.

**Examples**

The following SAS statements produce these results.

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>state1=zipnamel('27511'); put state1;</td>
<td>North Carolina</td>
</tr>
<tr>
<td>state2=zipnamel('01040'); put state2;</td>
<td>Massachusetts</td>
</tr>
<tr>
<td>state3=zipnamel(1040); put state3;</td>
<td>Massachusetts</td>
</tr>
<tr>
<td>state4=zipnamel(59017); put state4;</td>
<td>Montana</td>
</tr>
<tr>
<td>state5=zipnamel(24862); put state5;</td>
<td>West Virginia</td>
</tr>
</tbody>
</table>

**See Also**

Functions:
"ZIPCITY Function” on page 921
"ZIPFIPS Function” on page 922
"ZIPNAME Function” on page 924
"ZIPSTATE Function” on page 926

**ZIPSTATE Function**

Converts ZIP codes to two-character state postal codes
Functions and CALL Routines

ZIPSTATE Function

Category: State and ZIP Code

Syntax

ZIPSTATE(zip-code)

Arguments

zip-code
specifies a numeric or character expression that contains a five-digit ZIP code.

Tip: If the value of zip-code begins with leading zeros, you can enter the value without the leading zeroes. For example, if you enter 1040, ZIPSTATE assumes that the value is 01040.

Details

If the ZIPSTATE function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 20.

ZIPSTATE returns the two-character state postal code (or world-wide GSA geographic code for U.S. territories) that corresponds to its five-digit ZIP code argument. ZIPSTATE returns character values in uppercase.

Note: The first three digits of the ZIP code identifies a zone. ZIPSTATE evaluates the zone, and provides a state for that zone. It does not validate the ZIP code.

Comparisons

The ZIPCITY, ZIPNAME, ZIPNAMEL and ZIPSTATE functions take the same argument but return different values:

- ZIPCITY returns the uppercase name of the city and the two-character postal code that corresponds to its five-digit ZIP code argument.
- ZIPNAME returns the uppercase name of the state or U.S. territory that corresponds to its five-digit ZIP code argument.
- ZIPNAMEL returns the mixed case name of the state or U.S. territory that corresponds to its five-digit ZIP code argument.
- ZIPSTATE returns the uppercase two-character state postal code (or world-wide GSA geographic code for U.S. territories) that corresponds to its five-digit ZIP code argument.

Examples

The following SAS statements produce these results.

<table>
<thead>
<tr>
<th>SAS Statements</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>state1=zipstate(’27511’); put state1;</td>
<td>NC</td>
</tr>
<tr>
<td>state2=zipstate(’01040’); put state2;</td>
<td>MA</td>
</tr>
<tr>
<td>state3=zipstate(1040); put state3;</td>
<td>MA</td>
</tr>
</tbody>
</table>
SAS Statements Results

\begin{align*}
\text{state4}=\text{zipstate}(59017); \\
\text{put state4;} \\
\text{MT} \\
\text{state5}=\text{zipstate}(24862); \\
\text{put state5;} \\
\text{WV}
\end{align*}

See Also

Functions:

“ZIPFIPS Function” on page 922
“ZIPNAME Function” on page 924
“ZIPNAMEL Function” on page 925
“ZIPCITY Function” on page 921

References


Amos, D.E., Daniel, S.L., and Weston, K. (1977), “CDC 6600 Subroutines IBESS and JBESS for Bessel Functions $I(v,x)$ and $J(v,x)$, $x \geq 0, v \geq 0$,” ACM Transactions on Mathematical Software, 3, 76–95.


CHAPTER 5
Informats

Definition of Informats 932
Syntax 932
Using Informats 933
Ways to Specify Informats 933
INPUT Statement 933
INPUT Function 933
INFORMAT Statement 933
ATTRIB Statement 934
Permanent versus Temporary Association 934
User-Defined Informats 934

Byte Ordering for Integer Binary Data on Big Endian and Little Endian Platforms 935
Definitions 935
How the Bytes are Ordered 935
Reading Data Generated on Big Endian or Little Endian Platforms 936

Integer Binary Notation in Different Programming Languages 936

Working with Packed Decimal and Zoned Decimal Data 937
Definitions 937
Types of Data 937
Packed Decimal Data 937
Zoned Decimal Data 938
Packed Julian Dates 938

Platforms Supporting Packed Decimal and Zoned Decimal Data 938
Languages Supporting Packed Decimal and Zoned Decimal Data 938
Summary of Packed Decimal and Zoned Decimal Formats and Informats 939

Informats by Category 941
Dictionary 948
$ASCIIw. Informat 948
$BINARYw. Informat 949
$CBw. Informat 950
$CHARw. Informat 951
$CHARZBw. Informat 953
$CPTDWw. Informat 954
$CPTWDw. Informat 954
$EBCDICw. Informat 954
$HEXw. Informat 955
$KANJIw. Informat 956
$KANJIIXw. Informat 956
$LOGVSw. Informat 956
$LOGVSRw. Informat 956
$OCTALw. Informat 957
$PHEXw. Informat 958
$QUOTEw. Informat 959
$REVERJw. Informat 960
$REVERSw. Informat 960
$UCS2Bw. Informat 960
$UCS2Bew. Informat 960
$UCS2Lw. Informat 960
$UCS2Lex. Informat 961
$UCS2Xw. Informat 961
$UCS2Xew. Informat 961
$UCS4Bw. Informat 961
$UCS4Lw.d Informat 962
$UCS4Lex. Informat 962
$UCS4Xw. Informat 962
$UESCw. Informat 962
$UESCEw. Informat 963
$UNCRw. Informat 963
$UNCREw. Informat 963
$UPARENw. Informat 963
$UPARENw. Informat 963
$UPARENpw. Informat 964
$UPCASEw. Informat 964
$UTF8Xw. Informat 965
$VARYINGw. Informat 965
$VSLOGw. Informat 967
$VSLOGRw. Informat 967
$w. Informat 967
ANYDTDEw. Informat 968
ANYDTDTMw. Informat 970
ANYDTTMw. Informat 971
BINARYw.d Informat 973
BITSw.d Informat 974
BZw.d Informat 975
CBw.d Informat 976
COMMAw.d Informat 977
COMMAXw.d Informat 978
DATEw. Informat 979
DATETIMEw. Informat 980
DDMYYWw. Informat 982
Ew.d Informat 983
EURDFDEw. Informat 984
EURDFDTw. Informat 985
EURDFMYw. Informat 985
EUROw.d Informat 985
EUROXw.d Informat 985
FLOATw.d Informat 985
HEXw. Informat 987
IBw.d Informat 988
IBRw.d Informat 989
IEEEw.d Informat 990
JDATEYMDw. Informat 991
JNENGOw. Informat 992
JULIANw. Informat 992
MINGUOw. Informat 993
MMDDYYw. Informat 993
<table>
<thead>
<tr>
<th>Informats</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>MONYYw. Informat</td>
<td>995</td>
</tr>
<tr>
<td>MSECw. Informat</td>
<td>996</td>
</tr>
<tr>
<td>NENGOw. Informat</td>
<td>997</td>
</tr>
<tr>
<td>NLDATEw. Informat</td>
<td>998</td>
</tr>
<tr>
<td>NLDATMw. Informat</td>
<td>998</td>
</tr>
<tr>
<td>NLMNYw.d Informat</td>
<td>998</td>
</tr>
<tr>
<td>NLMNYw.d Informat</td>
<td>998</td>
</tr>
<tr>
<td>NLNUMw.d Informat</td>
<td>998</td>
</tr>
<tr>
<td>NLNUM1w.d Informat</td>
<td>998</td>
</tr>
<tr>
<td>NLPCTw.d Informat</td>
<td>999</td>
</tr>
<tr>
<td>NLPCT1w.d Informat</td>
<td>999</td>
</tr>
<tr>
<td>NLTMAPw. Informat</td>
<td>999</td>
</tr>
<tr>
<td>NLTIMEw. Informat</td>
<td>1000</td>
</tr>
<tr>
<td>NUMXw.d Informat</td>
<td>1000</td>
</tr>
<tr>
<td>OCTALw.d Informat</td>
<td>1001</td>
</tr>
<tr>
<td>PDw.d Informat</td>
<td>1002</td>
</tr>
<tr>
<td>PDJULGw. Informat</td>
<td>1003</td>
</tr>
<tr>
<td>PDJUL1w. Informat</td>
<td>1005</td>
</tr>
<tr>
<td>PDTIMEw. Informat</td>
<td>1006</td>
</tr>
<tr>
<td>PERCENTw.d Informat</td>
<td>1007</td>
</tr>
<tr>
<td>PIBw.d Informat</td>
<td>1008</td>
</tr>
<tr>
<td>PIBw.d Informat</td>
<td>1008</td>
</tr>
<tr>
<td>PKw.d Informat</td>
<td>1011</td>
</tr>
<tr>
<td>PUNCH.d Informat</td>
<td>1012</td>
</tr>
<tr>
<td>RBw.d Informat</td>
<td>1013</td>
</tr>
<tr>
<td>RMFDUw. Informat</td>
<td>1015</td>
</tr>
<tr>
<td>RMFDUw. Informat</td>
<td>1015</td>
</tr>
<tr>
<td>RMFSTAMPw. Informat</td>
<td>1016</td>
</tr>
<tr>
<td>ROWw.d Informat</td>
<td>1018</td>
</tr>
<tr>
<td>S370FFw.d Informat</td>
<td>1019</td>
</tr>
<tr>
<td>S370FIBw.d Informat</td>
<td>1020</td>
</tr>
<tr>
<td>S370FIBw.d Informat</td>
<td>1020</td>
</tr>
<tr>
<td>S370FPDw.d Informat</td>
<td>1023</td>
</tr>
<tr>
<td>S370FPDw.d Informat</td>
<td>1023</td>
</tr>
<tr>
<td>S370FIBUw.d Informat</td>
<td>1022</td>
</tr>
<tr>
<td>S370FIBUw.d Informat</td>
<td>1022</td>
</tr>
<tr>
<td>S370FZDw.d Informat</td>
<td>1028</td>
</tr>
<tr>
<td>S370FZDw.d Informat</td>
<td>1028</td>
</tr>
<tr>
<td>S370FZDLw.d Informat</td>
<td>1029</td>
</tr>
<tr>
<td>S370FZDLw.d Informat</td>
<td>1029</td>
</tr>
<tr>
<td>S370FZDSw.d Informat</td>
<td>1030</td>
</tr>
<tr>
<td>S370FZDSw.d Informat</td>
<td>1030</td>
</tr>
<tr>
<td>S370FZDTw.d Informat</td>
<td>1031</td>
</tr>
<tr>
<td>S370FZDTw.d Informat</td>
<td>1031</td>
</tr>
<tr>
<td>S370FZDUw.d Informat</td>
<td>1032</td>
</tr>
<tr>
<td>S370FZDUw.d Informat</td>
<td>1032</td>
</tr>
<tr>
<td>SHRSTAMPw. Informat</td>
<td>1033</td>
</tr>
<tr>
<td>SMFSTAMPw. Informat</td>
<td>1035</td>
</tr>
<tr>
<td>STIMEw. Informat</td>
<td>1036</td>
</tr>
<tr>
<td>STIMEw. Informat</td>
<td>1036</td>
</tr>
<tr>
<td>TODSTAMPw. Informat</td>
<td>1039</td>
</tr>
<tr>
<td>TRAILSGNw. Informat</td>
<td>1039</td>
</tr>
<tr>
<td>TUw. Informat</td>
<td>1040</td>
</tr>
<tr>
<td>VAXRBw.d Informat</td>
<td>1041</td>
</tr>
<tr>
<td>VAXRBw.d Informat</td>
<td>1041</td>
</tr>
<tr>
<td>WEEKUw. Informat</td>
<td>1042</td>
</tr>
<tr>
<td>WEEKUw. Informat</td>
<td>1042</td>
</tr>
<tr>
<td>WEEKVw. Informat</td>
<td>1043</td>
</tr>
<tr>
<td>WEEKVw. Informat</td>
<td>1043</td>
</tr>
<tr>
<td>WEEKWw. Informat</td>
<td>1044</td>
</tr>
<tr>
<td>WEEKWw. Informat</td>
<td>1044</td>
</tr>
<tr>
<td>YENw.d Informat</td>
<td>1044</td>
</tr>
<tr>
<td>YENw.d Informat</td>
<td>1044</td>
</tr>
<tr>
<td>YYMMDDw. Informat</td>
<td>1044</td>
</tr>
</tbody>
</table>
Definition of Informat

An informat is an instruction that SAS uses to read data values into a variable. For example, the following value contains a dollar sign and commas:

$1,000,000

To remove the dollar sign ($) and commas (,) before storing the numeric value 1000000 in a variable, read this value with the COMMA11. informat.

Unless you explicitly define a variable first, SAS uses the informat to determine whether the variable is numeric or character. SAS also uses the informat to determine the length of character variables.

Syntax

SAS informats have the following form:

<$$>informat<w><d>

where

$ indicates a character informat; its absence indicates a numeric informat.

informat names the informat. The informat is a SAS informat or a user-defined informat that was previously defined with the INVALUE statement in PROC FORMAT. For more information on user-defined informats, see the FORMAT procedure in the Base SAS Procedures Guide.

w specifies the informat width, which for most informats is the number of columns in the input data.

d specifies an optional decimal scaling factor in the numeric informats. SAS divides the input data by 10 to the power of d.

Note: Even though SAS can read up to 31 decimal places when you specify some numeric informats, floating-point numbers with more than 12 decimal places might lose precision due to the limitations of the eight-byte floating-point representation used by most computers.

Informat widths always contain a period (.) as a part of the name. If you omit the w and the d values from the informat, SAS uses default values. If the data contain decimal points, SAS ignores the d value and reads the number of decimal places that are actually in the input data.

If the informat width is too narrow to read all the columns in the input data, you may get unexpected results. The problem frequently occurs with the date and time
informats. You must adjust the width of the informat to include blanks or special characters between the day, month, year, or time. For more information about date and time values, see the discussion on SAS date and time values in SAS Language Reference: Concepts.

When a problem occurs with an informat, SAS writes a note to the SAS log and assigns a missing value to the variable. Problems occur if you use an incompatible informat, such as a numeric informat to read character data, or if you specify the width of a date and time informat that causes SAS to read a special character in the last column.

Using Informat

Ways to Specify Informat

You can specify informats in the following ways:

- in an INPUT statement
- with the INPUT, INPUTC, and INPUTN functions
- in an INFORMAT statement in a DATA step or a PROC step
- in an ATTRIB statement in a DATA step or a PROC step.

INPUT Statement

The INPUT statement with an informat after a variable name is the simplest way to read values into a variable. For example, the following INPUT statement uses two informats:

```
input @15 style $3. @21 price 5.2;
```

The $w. character informat reads values into the variable STYLE. The w.d numeric informat reads values into the variable PRICE.

For a complete discussion of the INPUT statement, see “INPUT Statement” on page 1248.

INPUT Function

The INPUT function reads a SAS character expression using a specified informat. The informat determines whether the resulting value is numeric or character. Thus, the INPUT function is useful for converting data. For example,

```
TempCharacter='98.6';
TemperatureNumber=input(TempCharacter,4.);
```

Here, the INPUT function in combination with the w.d informat reads the character value of TempCharacter as a numeric value and assigns the numeric value 98.6 to TemperatureNumber.

Use the PUT function with a SAS format to convert numeric values to character values. See “PUT Function” on page 754 for an example of a numeric-to-character conversion. For a complete discussion of the INPUT function, see “INPUT Function” on page 592.

INFORMAT Statement

The INFORMAT statement associates an informat with a variable. SAS uses the informat in any subsequent INPUT statement to read values into the variable. For
example, in the following statements the INFORMAT statement associates the DATEw. informat with the variables Birthdate and Interview:

\begin{verbatim}
   informat Birthdate Interview date9.;
   input @63 Birthdate Interview;
\end{verbatim}

An informat that is associated with an INFORMAT statement behaves like an informat that you specify with a colon (:) format modifier in an INPUT statement. (For details about using the colon (:) modifier, see the “INPUT Statement, List” on page 1269.) Therefore, SAS uses a modified list input to read the variable so that

- the \textit{w} value in an informat does not determine column positions or input field widths in an external file
- the blanks that are embedded in input data are treated as delimiters unless you change the \texttt{DELMITERS=} option in an INFILE statement
- for character informats, the \textit{w} value in an informat specifies the length of character variables
- for numeric informats, the \textit{w} value is ignored
- for numeric informats, the \textit{d} value in an informat behaves in the usual way for numeric informats.

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.

See “INPUT Statement, List” on page 1269 for more information on how to use modified list input to read data.

**ATTRIB Statement**

The ATTRIB statement can also associate an informat, as well as other attributes, with one or more variables. For example, in the following statements, the ATTRIB statement associates the DATEw. informat with the variables Birthdate and Interview:

\begin{verbatim}
   attrib Birthdate Interview informat=date9.;
   input @63 Birthdate Interview;
\end{verbatim}

An informat that is associated by using the \texttt{INFORMAT=} option in the ATTRIB statement behaves like an informat that you specify with a colon (:) format modifier in an INPUT statement. (For details about using the colon (:) modifier, see the “INPUT Statement, List” on page 1269.) Therefore, SAS uses a modified list input to read the variable in the same way as it does for the INFORMAT statement.

See “ATTRIB Statement” on page 1111 for more information.

**Permanent versus Temporary Association**

When you specify an informat in an INPUT statement, SAS uses the informat to read input data values during that DATA step. SAS, however, does not permanently associate the informat with the variable. To permanently associate a format with a variable, use an INFORMAT statement or an ATTRIB statement. SAS permanently associates an informat with the variable by modifying the descriptor information in the SAS data set.

**User-Defined Informats**

In addition to the informats that are supplied with Base SAS software, you can create your own informats. In Base SAS software, PROC FORMAT allows you to create
your own informats and formats for both character and numeric variables. For more information on user-defined informats, see the FORMAT procedure in the Base SAS Procedures Guide.

When you execute a SAS program that uses user-defined informats, these informats should be available. The two ways to make these informats available are

- to create permanent, not temporary, informats with PROC FORMAT
- to store the source code that creates the informats (the PROC FORMAT step) with the SAS program that uses them.

If you execute a program that cannot locate a user-defined informat, the result depends on the setting of the FMTERR= system option. If the user-defined informat is not found, then these system options produce these results:

<table>
<thead>
<tr>
<th>System Options</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMTERR</td>
<td>SAS produces an error that causes the current DATA or PROC step to stop.</td>
</tr>
<tr>
<td>NOFMTERR</td>
<td>SAS continues processing by substituting a default informat.</td>
</tr>
</tbody>
</table>

Although using NOFMTERR enables SAS to process a variable, you lose the information that the user-defined informat supplies. This option can cause a DATA step to misread data, and it can produce incorrect results.

To avoid problems, make sure that users of your program have access to all the user-defined informats that are used.

---

### Byte Ordering for Integer Binary Data on Big Endian and Little Endian Platforms

#### Definitions

Integer values for integer binary data are typically stored in one of three sizes: one–byte, two–byte, or four–byte. The ordering of the bytes for the integer varies depending on the platform (operating environment) on which the integers were produced.

The ordering of bytes differs between the “big endian” and the “little endian” platforms. These colloquial terms are used to describe byte ordering for IBM mainframes (big endian) and for Intel-based platforms (little endian). In the SAS System, the following platforms are considered big endian: IBM mainframe, HP-UX, AIX, Solaris, and Macintosh. The following platforms are considered little endian: OpenVMS Alpha, Digital UNIX, Intel ABI, and Windows.

#### How the Bytes are Ordered

On big endian platforms, the value 1 is stored in binary and is represented here in hexadecimal notation. One byte is stored as 01, two bytes as 00 01, and four bytes as 00 00 00 01. On little endian platforms, the value 1 is stored in one byte as 01 (the same as big endian), in two bytes as 01 00, and in four bytes as 01 00 00 00.
If an integer is negative, the “two’s complement” representation is used. The high-order bit of the most significant byte of the integer will be set on. For example, \(-2\) would be represented in one, two, and four bytes on big endian platforms as FE, FF FE, and FF FF FF FE respectively. On little endian platforms, the representation would be FE, FE FF, and FE FF FF FE. These representations result from the output of the integer binary value \(-2\) expressed in hexadecimal representation.

Reading Data Generated on Big Endian or Little Endian Platforms

SAS can read signed and unsigned integers regardless of whether they were generated on a big endian or a little endian system. Likewise, SAS can write signed and unsigned integers in both big endian and little endian format. The length of these integers can be up to eight bytes.

The following table shows which informat to use for various combinations of platforms. In the Sign? column, “no” indicates that the number is unsigned and cannot be negative. “Yes” indicates that the number can be either negative or positive.

Table 5.1  SAS Informats and Byte Ordering

<table>
<thead>
<tr>
<th>Data created for...</th>
<th>Data read on...</th>
<th>Sign?</th>
<th>Informat</th>
</tr>
</thead>
<tbody>
<tr>
<td>big endian</td>
<td>big endian</td>
<td>yes</td>
<td>IB or S370FIB</td>
</tr>
<tr>
<td>big endian</td>
<td>big endian</td>
<td>no</td>
<td>PIB, S370FPIB, S370FIBU</td>
</tr>
<tr>
<td>big endian</td>
<td>little endian</td>
<td>yes</td>
<td>IBR</td>
</tr>
<tr>
<td>big endian</td>
<td>little endian</td>
<td>no</td>
<td>PIBR</td>
</tr>
<tr>
<td>little endian</td>
<td>big endian</td>
<td>yes</td>
<td>IBR</td>
</tr>
<tr>
<td>little endian</td>
<td>big endian</td>
<td>no</td>
<td>PIBR</td>
</tr>
<tr>
<td>little endian</td>
<td>little endian</td>
<td>yes</td>
<td>IB or IBR</td>
</tr>
<tr>
<td>little endian</td>
<td>little endian</td>
<td>no</td>
<td>PIB or PIBR</td>
</tr>
<tr>
<td>big endian</td>
<td>either</td>
<td>yes</td>
<td>S370FIB</td>
</tr>
<tr>
<td>big endian</td>
<td>either</td>
<td>no</td>
<td>S370FPIB</td>
</tr>
<tr>
<td>little endian</td>
<td>either</td>
<td>yes</td>
<td>IB</td>
</tr>
<tr>
<td>little endian</td>
<td>either</td>
<td>no</td>
<td>PIBR</td>
</tr>
</tbody>
</table>

Integer Binary Notation in Different Programming Languages

The following table compares integer binary notation according to programming language.

Table 5.2  Integer Binary Notation and Programming Languages

<table>
<thead>
<tr>
<th>Language</th>
<th>2 Bytes</th>
<th>4 Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL/I</td>
<td>FIXED BIN(15)</td>
<td>FIXED BIN(31)</td>
</tr>
</tbody>
</table>
Working with Packed Decimal and Zoned Decimal Data

Definitions

Packed decimal specifies a method of encoding decimal numbers by using each byte to represent two decimal digits. Packed decimal representation stores decimal data with exact precision. The fractional part of the number is determined by the informat or format because there is no separate mantissa and exponent.

An advantage of using packed decimal data is that exact precision can be maintained. However, computations involving decimal data may become inexact due to the lack of native instructions.

Zoned decimal specifies a method of encoding decimal numbers in which each digit requires one byte of storage. The last byte contains the number's sign as well as the last digit. Zoned decimal data produces a printable representation.

Nibble specifies 1/2 of a byte.

Types of Data

Packed Decimal Data

A packed decimal representation stores decimal digits in each “nibble” of a byte. Each byte has two nibbles, and each nibble is indicated by a hexadecimal digit. For example, the value 15 is stored in two nibbles, using the hexadecimal digits 1 and 5.

The sign indication is dependent on your operating environment. On IBM mainframes, the sign is indicated by the last nibble. With formats, C indicates a positive value, and D indicates a negative value. With informats, A, C, E, and F indicate positive values, and B and D indicate negative values. Any other nibble is invalid for signed packed decimal data. In all other operating environments, the sign is indicated in its own byte. If the high-order bit is 1, then the number is negative. Otherwise, it is positive.

The following applies to packed decimal data representation:

- You can use the S370FPD format on all platforms to obtain the IBM mainframe configuration.
- You can have unsigned packed data with no sign indicator. The packed decimal format and informat handles the representation. It is consistent between ASCII and EBCDIC platforms.
Note that the S370FPDU format and informat expects to have an F in the last nibble, while packed decimal expects no sign nibble.

Zoned Decimal Data

The following applies to zoned decimal data representation:
- A zoned decimal representation stores a decimal digit in the low order nibble of each byte. For all but the byte containing the sign, the high-order nibble is the numeric zone nibble (F on EBCDIC and 3 on ASCII).
- The sign can be merged into a byte with a digit, or it can be separate, depending on the representation. But the standard zoned decimal format and informat expects the sign to be merged into the last byte.
- The EBCDIC and ASCII zoned decimal formats produce the same printable representation of numbers. There are two nibbles per byte, each indicated by a hexadecimal digit. For example, the value 15 is stored in two bytes. The first byte contains the hexadecimal value F1 and the second byte contains the hexadecimal value C5.

Packed Julian Dates

The following applies to packed Julian dates:
- The two formats and informats that handle Julian dates in packed decimal representation are PDJULJ and PDJULG. PDJULJ uses the IBM mainframe year computation, while PDJULG uses the Gregorian computation.
- The IBM mainframe computation considers 1900 to be the base year, and the year values in the data indicate the offset from 1900. For example, 98 means 1998, 100 means 2000, and 102 means 2002. 1998 would mean 3898.
- The Gregorian computation allows for 2–digit or 4–digit years. If you use 2–digit years, SAS uses the setting of the YEARCUTOFF= system option to determine the true year.

Platforms Supporting Packed Decimal and Zoned Decimal Data

Some platforms have native instructions to support packed and zoned decimal data, while others must use software to emulate the computations. For example, the IBM mainframe has an Add Pack instruction to add packed decimal data, but the Intel-based platforms have no such instruction and must convert the decimal data into some other format.

Languages Supporting Packed Decimal and Zoned Decimal Data

Several different languages support packed decimal and zoned decimal data. The following table shows how COBOL picture clauses correspond to SAS formats and informats.

<table>
<thead>
<tr>
<th>IBM VS COBOL II clauses</th>
<th>Corresponding S370Fxxx formats/informats</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC S9(X) PACKED-DECIMAL</td>
<td>S370FPDw.</td>
</tr>
<tr>
<td>PIC 9(X) PACKED-DECIMAL</td>
<td>S370FPDUw.</td>
</tr>
<tr>
<td>PIC S9(W) DISPLAY</td>
<td>S370FZDw.</td>
</tr>
</tbody>
</table>
IBM VS COBOL II clauses

| PIC 9(W) DISPLAY | S370FZDUw. |
| PIC S9(W) DISPLAY SIGN LEADING | S370FZDLw. |
| PIC S9(W) DISPLAY SIGN LEADING SEPARATE | S370FZDSw. |
| PIC S9(W) DISPLAY SIGN TRAILING SEPARATE | S370FZDTw. |

For the packed decimal representation listed above, X indicates the number of digits represented, and W is the number of bytes. For PIC S9(X) PACKED-DECIMAL, W is \( \text{ceil}(x/2) \). For PIC 9(X) PACKED-DECIMAL, W is \( \text{ceil}(x+1)/2 \). For example, PIC S9(5) PACKED-DECIMAL represents five digits. If a sign is included, six nibbles are needed. \( \text{ceil}(5+1)/2 \) has a length of three bytes, and the value of W is 3.

Note that you can substitute COMP-3 for PACKED-DECIMAL.

In IBM assembly language, the P directive indicates packed decimal, and the Z directive indicates zoned decimal. The following shows an excerpt from an assembly language listing, showing the offset, the value, and the DC statement:

```
offset value (in hex) inst label directive
+000000 00001C 2 PEX1 DC PL3'1'
+000003 00001D 3 PEX2 DC PL3'-1'
+000006 F0F0C1 4 ZEX1 DC ZL3'1'
+000009 F0F0D1 5 ZEX2 DC ZL3'1'
```

In PL/I, the FIXED DECIMAL attribute is used in conjunction with packed decimal data. You must use the PICTURE specification to represent zoned decimal data. There is no standardized representation of decimal data for the FORTRAN or the C languages.

**Summary of Packed Decimal and Zoned Decimal Formats and Informats**

SAS uses a group of formats and informats to handle packed and zoned decimal data. The following table lists the type of data representation for these formats and informats. Note that the formats and informats that begin with S370 refer to IBM mainframe representation.

<table>
<thead>
<tr>
<th>Format</th>
<th>Type of data representation</th>
<th>Corresponding informat</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>PD</td>
<td>Packed decimal</td>
<td>PD</td>
<td>Local signed packed decimal</td>
</tr>
<tr>
<td>PK</td>
<td>Packed decimal</td>
<td>PK</td>
<td>Unsigned packed decimal; not specific to your operating environment</td>
</tr>
<tr>
<td>ZD</td>
<td>Zoned decimal</td>
<td>ZD</td>
<td>Local zoned decimal</td>
</tr>
<tr>
<td>none</td>
<td>Zoned decimal</td>
<td>ZDB</td>
<td>Translates EBCDIC blank (hex 40) to EBCDIC zero (hex F0), then corresponds to the informat as zoned decimal</td>
</tr>
<tr>
<td>Format</td>
<td>Type of data representation</td>
<td>Corresponding informat</td>
<td>Comments</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------------</td>
<td>------------------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>none</td>
<td>Zoned decimal</td>
<td>ZDV</td>
<td>Non-IBM zoned decimal representation</td>
</tr>
<tr>
<td>S370FPD</td>
<td>Packed decimal</td>
<td>S370FPD</td>
<td>Last nibble C (positive) or D (negative)</td>
</tr>
<tr>
<td>S370FPDU</td>
<td>Packed decimal</td>
<td>S370FPDU</td>
<td>Last nibble always F (positive)</td>
</tr>
<tr>
<td>S370FZD</td>
<td>Zoned decimal</td>
<td>S370FZD</td>
<td>Last byte contains sign in upper nibble: C (positive) or D (negative)</td>
</tr>
<tr>
<td>S370FZDU</td>
<td>Zoned decimal</td>
<td>S370FZDU</td>
<td>Unsigned; sign nibble always F</td>
</tr>
<tr>
<td>S370FZDL</td>
<td>Zoned decimal</td>
<td>S370FZDL</td>
<td>Sign nibble in first byte in informat; separate leading sign byte of hex C0 (positive) or D0 (negative) in format</td>
</tr>
<tr>
<td>S370FZDS</td>
<td>Zoned decimal</td>
<td>S370FZDS</td>
<td>Leading sign of - (hex 60) or + (hex 4E)</td>
</tr>
<tr>
<td>S370FZDT</td>
<td>Zoned decimal</td>
<td>S370FZDT</td>
<td>Trailing sign of - (hex 60) or + (hex 4E)</td>
</tr>
<tr>
<td>PDJULI</td>
<td>Packed decimal</td>
<td>PDJULI</td>
<td>Julian date in packed representation - IBM computation</td>
</tr>
<tr>
<td>PDJULG</td>
<td>Packed decimal</td>
<td>PDJULG</td>
<td>Julian date in packed representation - Gregorian computation</td>
</tr>
<tr>
<td>none</td>
<td>Packed decimal</td>
<td>RMFDUR</td>
<td>Input layout is: mmsstttF</td>
</tr>
<tr>
<td>none</td>
<td>Packed decimal</td>
<td>SHRSTAMP</td>
<td>Input layout is: yyyydddFhhmmsssth, where yyyydddF is the packed Julian date; yyyy is a 0-based year from 1900</td>
</tr>
<tr>
<td>none</td>
<td>Packed decimal</td>
<td>SMFSTAMP</td>
<td>Input layout is: xxxxxxxxxxxyydddF, where yyyydddF is the packed Julian date; yyyy is a 0-based year from 1900</td>
</tr>
<tr>
<td>none</td>
<td>Packed decimal</td>
<td>PDTIME</td>
<td>Input layout is: 0hhmmssF</td>
</tr>
<tr>
<td>none</td>
<td>Packed decimal</td>
<td>RMFSTAMP</td>
<td>Input layout is: 0hhmmssFyyyydddF, where yyyydddF is the packed Julian date; yyyy is a 0-based year from 1900</td>
</tr>
</tbody>
</table>
Informats by Category

There are eight categories of informats in SAS:

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIDI text handling</td>
<td>instructs SAS to read bidirectional data values from data variables</td>
</tr>
<tr>
<td>Character</td>
<td>instructs SAS to read character data values into character variables.</td>
</tr>
<tr>
<td>Column Binary</td>
<td>instructs SAS to read data stored in column-binary or multipunched form into character and numeric variables.</td>
</tr>
<tr>
<td>DBCS</td>
<td>instructs SAS to handle various Asian languages.</td>
</tr>
<tr>
<td>Date and Time</td>
<td>instructs SAS to read data values into variables that represent dates, times, and datetimes.</td>
</tr>
<tr>
<td>Hebrew text handling</td>
<td>instructs SAS to read Hebrew data from data variables</td>
</tr>
<tr>
<td>Numeric</td>
<td>instructs SAS to read numeric data values into numeric variables.</td>
</tr>
<tr>
<td>User-Defined</td>
<td>instructs SAS to read data values by using an informat that is created with an INVALUE statement in PROC FORMAT.</td>
</tr>
</tbody>
</table>

For information on reading column-binary data, see SAS Language Reference: Concepts. For information on creating user-defined informats, see the FORMAT procedure in the Base SAS Procedures Guide.

The following table provides brief descriptions of the SAS informats. For more detailed descriptions, see the dictionary entry for each informat.

Table 5.3 Categories and Descriptions of Informats

<table>
<thead>
<tr>
<th>Category</th>
<th>Informats</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIDI text handling</td>
<td>“$LOGVS{w. Informat}” on page 956</td>
<td>Reads a character string that is in left-to-right logical order and then converts the character string to visual order</td>
</tr>
<tr>
<td></td>
<td>“$LOGVSR{w. Informat}” on page 956</td>
<td>Reads a character string that is in right-to-left logical order and then converts the character string to visual order</td>
</tr>
<tr>
<td></td>
<td>“$VSLOG{w. Informat}” on page 967</td>
<td>Reads a character string that is in visual order and then converts the character string to left-to-right logical order</td>
</tr>
<tr>
<td></td>
<td>“$VSLOGR{w. Informat}” on page 967</td>
<td>Reads a character string that is in visual order and then converts the character string to right-to-left logical order</td>
</tr>
<tr>
<td>Character</td>
<td>“$ASCII{w. Informat}” on page 948</td>
<td>Converts ASCII character data to native format</td>
</tr>
<tr>
<td></td>
<td>“$BINARY{w. Informat}” on page 949</td>
<td>Converts binary data to character data</td>
</tr>
<tr>
<td></td>
<td>“$CHAR{w. Informat}” on page 951</td>
<td>Reads character data with blanks</td>
</tr>
<tr>
<td></td>
<td>“$CHARZ{w. Informat}” on page 953</td>
<td>Converts binary 0s to blanks</td>
</tr>
<tr>
<td>Category</td>
<td>Informats</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>“$EBCDICw. Informat” on page 954</td>
<td>Converts EBCDIC character data to native format</td>
<td></td>
</tr>
<tr>
<td>“$HEXw. Informat” on page 955</td>
<td>Converts hexadecimal data to character data</td>
<td></td>
</tr>
<tr>
<td>“$OCTALw. Informat” on page 957</td>
<td>Converts octal data to character data</td>
<td></td>
</tr>
<tr>
<td>“$PHEXw. Informat” on page 958</td>
<td>Converts packed hexadecimal data to character data</td>
<td></td>
</tr>
<tr>
<td>“$QUOTEw. Informat” on page 959</td>
<td>Removes matching quotation marks from character data</td>
<td></td>
</tr>
<tr>
<td>“$REVERJw. Informat” on page 960</td>
<td>Reads character data from right to left and preserves blanks</td>
<td></td>
</tr>
<tr>
<td>“$REVERSww. Informat” on page 960</td>
<td>Reads character data from right to left and left, and then left aligns the text</td>
<td></td>
</tr>
<tr>
<td>“$UCS2Bw. Informat” on page 960</td>
<td>Reads a character string that is encoded in big-endian, 16-bit, universal character set code in 2 octets (UCS2), Unicode encoding, and then converts the character string to the encoding of the current SAS session</td>
<td></td>
</tr>
<tr>
<td>“$UCS2BEw. Informat” on page 960</td>
<td>Reads a character string that is in the encoding of the current SAS session and then converts the character string to big-endian, 16-bit, universal character set code in 2 octets (UCS2), Unicode encoding</td>
<td></td>
</tr>
<tr>
<td>“$UCS2Lw. Informat” on page 960</td>
<td>Reads a character string that is encoded in little-endian, 16-bit, universal character set code in 2 octets (UCS2), Unicode encoding, and then converts the character string to the encoding of the current SAS session</td>
<td></td>
</tr>
<tr>
<td>“$UCS2LEw. Informat” on page 961</td>
<td>Reads a character string that is in the encoding of the current SAS session and then converts the character string to little-endian, 16-bit, universal character set code in 2 octets (UCS2), Unicode encoding</td>
<td></td>
</tr>
<tr>
<td>“$UCS2Xw. Informat” on page 961</td>
<td>Reads a character string that is encoded in 16-bit, universal character set code in 2 octets (UCS2), Unicode encoding, and then converts the character string to the encoding of the current SAS session</td>
<td></td>
</tr>
<tr>
<td>“$UCS2XEw. Informat” on page 961</td>
<td>Reads a character string that is in the encoding of the current SAS session and then converts the character string to 16-bit, universal character set code in 2 octets (UCS2), Unicode encoding</td>
<td></td>
</tr>
<tr>
<td>“$UCS4Bw. Informat” on page 961</td>
<td>Reads a character string that is encoded in big-endian, 32-bit, universal character set code in 4 octets (UCS4), Unicode encoding, and then converts the character string to the encoding of the current SAS session</td>
<td></td>
</tr>
<tr>
<td>“$UCS4Lw.d Informat” on page 962</td>
<td>Reads a character string that is encoded in little-endian, 32-bit, universal character set code in 4 octets (UCS4), Unicode encoding, and then converts the character string to the encoding of the current SAS session</td>
<td></td>
</tr>
<tr>
<td>Category</td>
<td>Informat</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>---------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>&quot;$UCS4Xw. Informat&quot; on</td>
<td>Reads a character string that is encoded in 32-bit, universal character set code in 4 octets (UCS4), Unicode encoding, and then converts the</td>
</tr>
<tr>
<td></td>
<td>page 962</td>
<td>character string to the encoding of the current SAS session</td>
</tr>
<tr>
<td></td>
<td>&quot;$UCS4XEw. Informat&quot; on</td>
<td>Reads a character string that is in the encoding of the current SAS session and then converts the character string to 32-bit, universal</td>
</tr>
<tr>
<td></td>
<td>page 962</td>
<td>character set code in 4 octets (UCS4), Unicode encoding</td>
</tr>
<tr>
<td></td>
<td>&quot;$UESCw. Informat&quot; on</td>
<td>Reads a character string that is encoded in Unicode escape (UESC) representation, and then converts the character string to the encoding</td>
</tr>
<tr>
<td></td>
<td>page 962</td>
<td>of the current SAS session</td>
</tr>
<tr>
<td></td>
<td>&quot;$UESCEw. Informat&quot; on</td>
<td>Reads a character string that is in the encoding of the current SAS session, and then converts the character string to Unicode escape (UESC)</td>
</tr>
<tr>
<td></td>
<td>page 963</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;$UNCRw. Informat&quot; on</td>
<td>Reads the numeric character representation (NCR) character string, and then converts the character string to the encoding of the current SAS</td>
</tr>
<tr>
<td></td>
<td>page 963</td>
<td>session</td>
</tr>
<tr>
<td></td>
<td>&quot;$UNCREw. Informat&quot; on</td>
<td>Reads a character string in the encoding of the current SAS session, and then converts the character string to session-encoded NCR (numeric</td>
</tr>
<tr>
<td></td>
<td>page 963</td>
<td>character representation)</td>
</tr>
<tr>
<td></td>
<td>&quot;$UPARENw. Informat&quot; on</td>
<td>Reads a character string that is encoded in UPAREN (Unicode parenthesis) representation, and then converts the character string to the</td>
</tr>
<tr>
<td></td>
<td>page 963</td>
<td>encoding of the current SAS session</td>
</tr>
<tr>
<td></td>
<td>&quot;$UPARENEw. Informat&quot; on</td>
<td>Reads a character string that is encoded in Unicode parenthesis (UPAREN) representation, and then converts the character string to the</td>
</tr>
<tr>
<td></td>
<td>page 963</td>
<td>encoding of the current SAS session with national characters remaining in the encoding of the UPAREN representation</td>
</tr>
<tr>
<td></td>
<td>&quot;$UPCASEw. Informat&quot; on</td>
<td>Converts character data to uppercase</td>
</tr>
<tr>
<td></td>
<td>page 964</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;$UTF8Xw. Informat&quot; on</td>
<td>Reads a character string that is encoded in Unicode transformation format (UTF-8), and then converts the character string to the encoding</td>
</tr>
<tr>
<td></td>
<td>page 965</td>
<td>of the current SAS session</td>
</tr>
<tr>
<td></td>
<td>&quot;$VARYINGw. Informat&quot; on</td>
<td>Reads character data of varying length</td>
</tr>
<tr>
<td></td>
<td>page 965</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;$w. Informat&quot; on page</td>
<td>Reads standard character data</td>
</tr>
<tr>
<td></td>
<td>967</td>
<td></td>
</tr>
<tr>
<td>Column Binary</td>
<td>&quot;$SCBw. Informat&quot; on</td>
<td>Reads standard character data from column-binary files</td>
</tr>
<tr>
<td></td>
<td>page 950</td>
<td></td>
</tr>
<tr>
<td>Category</td>
<td>Informats</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>“CBw.d Informat” on page 976</td>
<td>“CBw.d Informat” on page 976</td>
<td>Reads standard numeric values from column-binary files</td>
</tr>
<tr>
<td>“PUNCH.d Informat” on page 1012</td>
<td>“PUNCH.d Informat” on page 1012</td>
<td>Reads whether a row of column-binary data is punched</td>
</tr>
<tr>
<td>“ROWw.d Informat” on page 1018</td>
<td>“ROWw.d Informat” on page 1018</td>
<td>Reads a column-binary field down a card column</td>
</tr>
<tr>
<td>DBCS</td>
<td>“$KANJIw. Informat” on page 956</td>
<td>“$KANJIw. Informat” on page 956</td>
</tr>
<tr>
<td></td>
<td>“$KANJIXw. Informat” on page 956</td>
<td>“$KANJIXw. Informat” on page 956</td>
</tr>
<tr>
<td>Date and Time</td>
<td>“ANYDTDTEw. Informat” on page 968</td>
<td>“ANYDTDTEw. Informat” on page 968</td>
</tr>
<tr>
<td></td>
<td>“ANYDTDTMw. Informat” on page 970</td>
<td>“ANYDTDTMw. Informat” on page 970</td>
</tr>
<tr>
<td></td>
<td>“ANYDTTMEw. Informat” on page 971</td>
<td>“ANYDTTMEw. Informat” on page 971</td>
</tr>
<tr>
<td></td>
<td>“DATEw. Informat” on page 979</td>
<td>“DATEw. Informat” on page 979</td>
</tr>
<tr>
<td></td>
<td>“DATETIMEw. Informat” on page 980</td>
<td>“DATETIMEw. Informat” on page 980</td>
</tr>
<tr>
<td></td>
<td>“DDMMYYw. Informat” on page 982</td>
<td>“DDMMYYw. Informat” on page 982</td>
</tr>
<tr>
<td></td>
<td>“EURDFDEw. Informat” on page 984</td>
<td>“EURDFDEw. Informat” on page 984</td>
</tr>
<tr>
<td></td>
<td>“EURDFDTw. Informat” on page 985</td>
<td>“EURDFDTw. Informat” on page 985</td>
</tr>
<tr>
<td></td>
<td>“EURDFMYw. Informat” on page 985</td>
<td>“EURDFMYw. Informat” on page 985</td>
</tr>
<tr>
<td></td>
<td>“JDATEYMDw. Informat” on page 991</td>
<td>“JDATEYMDw. Informat” on page 991</td>
</tr>
<tr>
<td></td>
<td>“JNENGOw. Informat” on page 992</td>
<td>“JNENGOw. Informat” on page 992</td>
</tr>
<tr>
<td></td>
<td>“JULIANw. Informat” on page 992</td>
<td>“JULIANw. Informat” on page 992</td>
</tr>
<tr>
<td></td>
<td>“MINGUOw. Informat” on page 993</td>
<td>“MINGUOw. Informat” on page 993</td>
</tr>
<tr>
<td></td>
<td>“MMDDYYw. Informat” on page 993</td>
<td>“MMDDYYw. Informat” on page 993</td>
</tr>
<tr>
<td></td>
<td>“MONYYw. Informat” on page 995</td>
<td>“MONYYw. Informat” on page 995</td>
</tr>
<tr>
<td>Category</td>
<td>Informs</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
<td></td>
</tr>
<tr>
<td>“MSECw. Informat” on page 996</td>
<td>“MSECw. Informat” on page 996 Reads TIME MIC values</td>
<td></td>
</tr>
<tr>
<td>“NENGOw. Informat” on page 997</td>
<td>“NENGOw. Informat” on page 997 Reads Japanese date values in the form eyymmdd</td>
<td></td>
</tr>
<tr>
<td>“NLDATEw. Informat” on page 998</td>
<td>“NLDATEw. Informat” on page 998 Reads the date value in the specified locale and then converts the date value to the local SAS date value</td>
<td></td>
</tr>
<tr>
<td>“NLDATMw. Informat” on page 998</td>
<td>“NLDATMw. Informat” on page 998 Reads the datetime value of the specified locale, and then converts the datetime value to the local SAS datetime value</td>
<td></td>
</tr>
<tr>
<td>“NLTIMAPw. Informat” on page 999</td>
<td>“NLTIMAPw. Informat” on page 999 Reads the time value and uses a.m. and p.m. in the specified locale, and then converts the time value to the local SAS-time value</td>
<td></td>
</tr>
<tr>
<td>“NLTIMEw. Informat” on page 1000</td>
<td>“NLTIMEw. Informat” on page 1000 Reads the time value in the specified locale and then converts the time value to the local SAS time value</td>
<td></td>
</tr>
<tr>
<td>“PDJULGw. Informat” on page 1003</td>
<td>“PDJULGw. Informat” on page 1003 Reads packed Julian date values in the hexadecimal form yyyydddF for IBM</td>
<td></td>
</tr>
<tr>
<td>“PDJULlw. Informat” on page 1005</td>
<td>“PDJULlw. Informat” on page 1005 Reads packed Julian dates in the hexadecimal format ccyyddd F for IBM</td>
<td></td>
</tr>
<tr>
<td>“PDTIMEw. Informat” on page 1006</td>
<td>“PDTIMEw. Informat” on page 1006 Reads packed decimal time of SMF and RMF records</td>
<td></td>
</tr>
<tr>
<td>“RMFDURw. Informat” on page 1015</td>
<td>“RMFDURw. Informat” on page 1015 Reads duration intervals of RMF records</td>
<td></td>
</tr>
<tr>
<td>“RMFSTAMPw. Informat” on page 1016</td>
<td>“RMFSTAMPw. Informat” on page 1016 Reads time and date fields of RMF records</td>
<td></td>
</tr>
<tr>
<td>“SHRSTAMPw. Informat” on page 1033</td>
<td>“SHRSTAMPw. Informat” on page 1033 Reads date and time values of SHR records</td>
<td></td>
</tr>
<tr>
<td>“SMFSTAMPw. Informat” on page 1035</td>
<td>“SMFSTAMPw. Informat” on page 1035 Reads time and date values of SMF records</td>
<td></td>
</tr>
<tr>
<td>“STIMERw. Informat” on page 1036</td>
<td>“STIMERw. Informat” on page 1036 Reads time values and determines whether the values are hours, minutes, or seconds; reads the output of the STIMER system option</td>
<td></td>
</tr>
<tr>
<td>“TIMEw. Informat” on page 1037</td>
<td>“TIMEw. Informat” on page 1037 Reads hours, minutes, and seconds in the form hh:mm:ss.ss</td>
<td></td>
</tr>
<tr>
<td>“TODSTAMPw. Informat” on page 1039</td>
<td>“TODSTAMPw. Informat” on page 1039 Reads an eight-byte time-of-day stamp</td>
<td></td>
</tr>
<tr>
<td>“TUw. Informat” on page 1040</td>
<td>“TUw. Informat” on page 1040 Reads timer units</td>
<td></td>
</tr>
<tr>
<td>“WEEKUw. Informat” on page 1043</td>
<td>“WEEKUw. Informat” on page 1043 Reads the format of the number-of-week value within the year and returns a SAS date value by using the U algorithm</td>
<td></td>
</tr>
<tr>
<td>“WEEKVw. Informat” on page 1043</td>
<td>“WEEKVw. Informat” on page 1043 Reads the format of the number-of-week value within the year and returns a SAS date value using the V algorithm</td>
<td></td>
</tr>
<tr>
<td>“WEEKWw. Informat” on page 1044</td>
<td>“WEEKWw. Informat” on page 1044 Reads the format of the number-of-week value within the year and returns a SAS date value using the W algorithm</td>
<td></td>
</tr>
<tr>
<td>Category</td>
<td>Informats</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>“YYMMDw. Informat” on page 1044</td>
<td>Reads date values in the form $yymdd$ or $yyymmdd$</td>
</tr>
<tr>
<td></td>
<td>“YYMMNw. Informat” on page 1045</td>
<td>Reads date values in the form $yyyymm$ or $yymm$</td>
</tr>
<tr>
<td></td>
<td>“YYQw. Informat” on page 1047</td>
<td>Reads quarters of the year in the form $yyQq$ or $yyyyQq$</td>
</tr>
<tr>
<td>Hebrew text handling</td>
<td>“$CPTDWw. Informat” on page 954</td>
<td>Reads a character string that is encoded in Hebrew DOS (cp862) and then converts the character string to Windows (cp1255) encoding</td>
</tr>
<tr>
<td></td>
<td>“$CPTWDw. Informat” on page 954</td>
<td>Reads a character string that is encoded in Windows (cp1255) and then converts the character string to Hebrew DOS (cp862) encoding</td>
</tr>
<tr>
<td>Numeric</td>
<td>“BINARYw.d Informat” on page 973</td>
<td>Converts positive binary values to integers</td>
</tr>
<tr>
<td></td>
<td>“BITSw.d Informat” on page 974</td>
<td>Extracts bits</td>
</tr>
<tr>
<td></td>
<td>“BZw.d Informat” on page 975</td>
<td>Converts blanks to 0s</td>
</tr>
<tr>
<td></td>
<td>“COMMAw.d Informat” on page 977</td>
<td>Removes embedded characters</td>
</tr>
<tr>
<td></td>
<td>“COMMAXw.d Informat” on page 978</td>
<td>Removes embedded characters</td>
</tr>
<tr>
<td></td>
<td>“Ew.d Informat” on page 983</td>
<td>Reads numeric values that are stored in scientific notation and double-precision scientific notation</td>
</tr>
<tr>
<td></td>
<td>“EUROw.d Informat” on page 985</td>
<td>Reads numeric values and removes embedded characters in European currency and reverses the comma and decimal point</td>
</tr>
<tr>
<td></td>
<td>“EUROXw.d Informat” on page 985</td>
<td>Reads numeric values and removes embedded characters in European currency</td>
</tr>
<tr>
<td></td>
<td>“FLOATw.d Informat” on page 985</td>
<td>Reads a native single-precision, floating-point value and divides it by 10 raised to the $d$th power</td>
</tr>
<tr>
<td></td>
<td>“HEXw. Informat” on page 987</td>
<td>Converts hexadecimal positive binary values to either integer (fixed-point) or real (floating-point) binary values</td>
</tr>
<tr>
<td></td>
<td>“IBw.d Informat” on page 988</td>
<td>Reads native integer binary (fixed-point) values, including negative values</td>
</tr>
<tr>
<td></td>
<td>“IBRw.d Informat” on page 989</td>
<td>Reads integer binary (fixed-point) values in Intel and DEC formats</td>
</tr>
<tr>
<td></td>
<td>“IEEEw.d Informat” on page 990</td>
<td>Reads an IEEE floating-point value and divides it by 10 raised to the $d$th power</td>
</tr>
<tr>
<td></td>
<td>“NLMNYw.d Informat” on page 998</td>
<td>Reads monetary data in the specified locale for the local expression, and converts the data to a numeric value</td>
</tr>
<tr>
<td></td>
<td>“NLMNYIw.d Informat” on page 998</td>
<td>Reads monetary data in the specified locale for the international expression, and then converts the data to a numeric value</td>
</tr>
<tr>
<td>Category</td>
<td>Informats</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>“NLNUMw.d Informat” on page 998</td>
<td>Reads numeric data in the specified locale for local expressions, and then converts the data to a numeric value</td>
</tr>
<tr>
<td></td>
<td>“NLNUMIw.d Informat” on page 999</td>
<td>Reads numeric data in the specified locale for international expressions, and then converts the data to a numeric value</td>
</tr>
<tr>
<td></td>
<td>“NLPCTw.d Informat” on page 999</td>
<td>Reads percentage data in the specified locale for local expressions, and then converts the data to a numeric value</td>
</tr>
<tr>
<td></td>
<td>“NLPCTIw.d Informat” on page 999</td>
<td>Reads percentage data in the specified locale for international expressions, and then converts the data to a numeric value</td>
</tr>
<tr>
<td></td>
<td>“NUMXw.d Informat” on page 1000</td>
<td>Reads numeric values with a comma in place of the decimal point</td>
</tr>
<tr>
<td></td>
<td>“OCTALw.d Informat” on page 1001</td>
<td>Converts positive octal values to integers</td>
</tr>
<tr>
<td></td>
<td>“PDw.d Informat” on page 1002</td>
<td>Reads data that are stored in IBM packed decimal format</td>
</tr>
<tr>
<td></td>
<td>“PERCENTTw.d Informat” on page 1007</td>
<td>Reads percentages as numeric values</td>
</tr>
<tr>
<td></td>
<td>“PIBw.d Informat” on page 1008</td>
<td>Reads positive integer binary (fixed-point) values</td>
</tr>
<tr>
<td></td>
<td>“PIBRe.w.d Informat” on page 1010</td>
<td>Reads positive integer binary (fixed-point) values in Intel and DEC formats</td>
</tr>
<tr>
<td></td>
<td>“PKw.d Informat” on page 1011</td>
<td>Reads unsigned packed decimal data</td>
</tr>
<tr>
<td></td>
<td>“RBw.d Informat” on page 1013</td>
<td>Reads numeric data that are stored in real binary (floating-point) notation</td>
</tr>
<tr>
<td></td>
<td>“S370FFw.d Informat” on page 1019</td>
<td>Reads EBCDIC numeric data</td>
</tr>
<tr>
<td></td>
<td>“S370FIBw.d Informat” on page 1020</td>
<td>Reads integer binary (fixed-point) values, including negative values, in IBM mainframe format</td>
</tr>
<tr>
<td></td>
<td>“S370FIBUw.d Informat” on page 1022</td>
<td>Reads unsigned integer binary (fixed-point) values in IBM mainframe format</td>
</tr>
<tr>
<td></td>
<td>“S370FP Dw.d Informat” on page 1023</td>
<td>Reads packed data in IBM mainframe format</td>
</tr>
<tr>
<td></td>
<td>“S370FPDUw.d Informat” on page 1024</td>
<td>Reads unsigned packed decimal data in IBM mainframe format</td>
</tr>
<tr>
<td></td>
<td>“S370FPIBw.d Informat” on page 1025</td>
<td>Reads positive integer binary (fixed-point) values in IBM mainframe format</td>
</tr>
<tr>
<td></td>
<td>“S370FRBw.d Informat” on page 1027</td>
<td>Reads real binary (floating-point) data in IBM mainframe format</td>
</tr>
<tr>
<td></td>
<td>“S370FZDw.d Informat” on page 1028</td>
<td>Reads zoned decimal data in IBM mainframe format</td>
</tr>
<tr>
<td>Category</td>
<td>Informats</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td>“S370FZDLw.d Informat” on page 1029</td>
<td>Reads zoned decimal leading-sign data in IBM mainframe format</td>
</tr>
<tr>
<td></td>
<td>“S370FZDSw.d Informat” on page 1030</td>
<td>Reads zoned decimal separate leading-sign data in IBM mainframe format</td>
</tr>
<tr>
<td></td>
<td>“S370FZDTw.d Informat” on page 1031</td>
<td>Reads zoned decimal separate trailing-sign data in IBM mainframe format</td>
</tr>
<tr>
<td></td>
<td>“S370FZDUw.d Informat” on page 1032</td>
<td>Reads unsigned zoned decimal data in IBM mainframe format</td>
</tr>
<tr>
<td></td>
<td>“TRAILSGNW. Informat” on page 1039</td>
<td>Reads a trailing plus (+) or minus (-) sign</td>
</tr>
<tr>
<td></td>
<td>“VAXRbw.d Informat” on page 1041</td>
<td>Reads real binary (floating-point) data in VMS format</td>
</tr>
<tr>
<td></td>
<td>“w.d Informat” on page 1042</td>
<td>Reads standard numeric data</td>
</tr>
<tr>
<td></td>
<td>“YENw.d Informat” on page 1044</td>
<td>Removes embedded yen signs, commas, and decimal points</td>
</tr>
<tr>
<td></td>
<td>“ZDw.d Informat” on page 1048</td>
<td>Reads zoned decimal data</td>
</tr>
<tr>
<td></td>
<td>“ZDBw.d Informat” on page 1050</td>
<td>Reads zoned decimal data in which zeros have been left blank</td>
</tr>
<tr>
<td></td>
<td>“ZDVw.d Informat” on page 1050</td>
<td>Reads and validates zoned decimal data</td>
</tr>
</tbody>
</table>

**Dictionary**

\textbf{$\$\text{ASCII}\, w.\, \text{Informat}$}

Converts ASCII character data to native format

**Category:** Character

**Syntax**

\$\text{ASCII}\, w.$

**Syntax Description**

\(w\)

specifies the width of the input field.

**Default:** 1 if the length of the variable is undefined; otherwise, the length of the variable
**Range:** 1–32767

**Details**
If ASCII is the native format, no conversion occurs.

**Comparisons**
- On an IBM mainframe system, $\text{ASCII}_w$. converts ASCII data to EBCDIC.
- On all other systems, $\text{ASCII}_w$. behaves like the $\text{CHAR}_w$. informat except that the default length is different.

**Examples**

```plaintext
input @1 name $\text{ascii3.};
```

<table>
<thead>
<tr>
<th>When the Data Line = ...</th>
<th>EBCDIC</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc</td>
<td>818283</td>
<td>616263</td>
</tr>
<tr>
<td>ABC</td>
<td>C1C2C3</td>
<td>414243</td>
</tr>
<tr>
<td>()</td>
<td>4D5D5E</td>
<td>28293B</td>
</tr>
</tbody>
</table>

* The results are hexadecimal representations of codes for characters. Each two hexadecimal digits correspond to one byte of binary data, and each byte corresponds to one character value.

---

**$\text{BINARY}_w$. Informat**

Converts binary data to character data

**Category:** Character

**Syntax**

$\text{BINARY}_w$.

**Syntax Description**

$w$

specifies the width of the input field. Because eight bits of binary information represent one character, every eight characters of input that $\text{BINARY}_w$. reads becomes one character value stored in a variable.

If $w<8$, $\text{BINARY}_w$. reads the data as $w$ characters followed by 0s. Thus, $\text{BINARY}_4$. reads the characters 0101 as 01010000, which converts to an EBCDIC & or an ASCII P. If $w>8$ but is not a multiple of 8, $\text{BINARY}_w$. reads up to the largest multiple of 8 that is less than $w$ before converting the data.
Details
The $BINARYw.$ informat does not interpret actual binary data, but it converts a string of characters that contains only 0s or 1s as though it is actual binary information. Therefore, use only the character digits 1 and 0 in the input, with no embedded blanks. $BINARYw.$ ignores leading and trailing blanks.

To read representations of binary codes for unprintable characters, enter an ASCII or EBCDIC equivalent for a particular character as a string of 0s and 1s. The $BINARYw.$ informat converts the string to its equivalent character value.

Comparisons
- The BINARYw. informat reads eight characters of input that contain only 0s or 1s as a binary representation of one byte of numeric data.
- The $HEXw.$ informat reads hexadecimal digits that represent the ASCII or EBCDIC equivalent of character data.

Examples

```
input @1 name $binary16.;
```

| When the Data Line = ... | The Result is ...
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-----+-----+-----+-----+-----+-----+-----+-----+-----</td>
<td>ASCII</td>
</tr>
<tr>
<td>0100110001001101</td>
<td>LM</td>
</tr>
</tbody>
</table>

$CBw.$ Informat

Reads standard character data from column-binary files

Syntax
$CBw.$

Syntax Description

$w$
specifies the width of the input field.

Default: none
Range: 1–32767
Details

The $CBw. informat reads standard character data from column-binary files, with each card column represented in 2 bytes, and it translates the data into standard character codes. If the combinations are invalid punch codes, SAS returns blanks and sets the automatic variable _ERROR_ to 1.

Examples

```
input @1 name $cb2.;
```

<table>
<thead>
<tr>
<th>Data Line* = ...</th>
<th>EBCDIC</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>200A</td>
<td>+</td>
<td>N</td>
</tr>
</tbody>
</table>

* The data line is a hexadecimal representation of the column binary. The punch card column for the example data has row 12, row 6, and row 8 punched. The binary representation is 0010 0000 0000 1010.

See Also

Informats:
- “CBw.d Informat” on page 976
- “PUNCH.d Informat” on page 1012
- “ROWw.d Informat” on page 1018

See the discussion on reading column-binary data in SAS Language Reference: Concepts.

$CHARw. Informat

Reads character data with blanks

Category: Character

Syntax

$CHARw.

Syntax Description

$w

specifies the width of the input field.
Default: 8 if the length of the variable is undefined; otherwise, the length of the variable

Range: 1–32767

Details

The $CHARw. informat does not trim leading and trailing blanks or convert a single period in the input data field to a blank before storing values. If you use $CHARw. in an INFORMAT or ATTRIB statement within a DATA step to read list input, then by default SAS interprets any blank embedded within data as a field delimiter, including leading blanks.

Comparisons

☐ The $CHARw. informat is almost identical to the $w. informat. However $CHARw. does not trim leading blanks or convert a single period in the input data field to a blank, while the $w. informat does.

☐ Use the table below to compare the SAS informat $CHAR8. with notation in other programming languages:

<table>
<thead>
<tr>
<th>Language</th>
<th>Character Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAS</td>
<td>$CHAR8.</td>
</tr>
<tr>
<td>IBM 370 assembler</td>
<td>CL8</td>
</tr>
<tr>
<td>C</td>
<td>char [8]</td>
</tr>
<tr>
<td>COBOL</td>
<td>PIC x(8)</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>A8</td>
</tr>
<tr>
<td>PL/I</td>
<td>CHAR(8)</td>
</tr>
</tbody>
</table>

Examples

```
input @1 name $char5.;
```

When the Data Line = ...
The Result* is ...

| ----+----1 |
| ------+-----|
| XYZ   | XYZ## |
| XYZ   | #XYZ# |
| .     | ##.## |
| X YZ  | #X#YZ |

* The character # represents a blank space.
$\text{CHARZB}_w$. Informat

Converts binary 0s to blanks

Category: Character

Syntax

$\text{CHARZB}_w$.

Syntax Description

\(w\)

specifies the width of the input field.

Default: 1 if the length of the variable is undefined; otherwise, the length of the variable

Range: 1–32767

Details

The $\text{CHARZB}_w$. informat does not trim leading and trailing blanks in character data before it stores values.

Comparisons

The $\text{CHARZB}_w$. informat is identical to the $\text{CHAR}_w$. informat except that $\text{CHARZB}_w$. converts any byte that contains a binary 0 to a blank character.

Examples

```
input @1 name $charzb5.;
```

<table>
<thead>
<tr>
<th>When the Data Line* = ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>E7E8E90000</td>
<td>58595A0000</td>
</tr>
<tr>
<td>00E7E8E900</td>
<td>0058595A00</td>
</tr>
<tr>
<td>00E700E8E9</td>
<td>005800595A</td>
</tr>
</tbody>
</table>

* The data lines are hexadecimal representations of codes for characters. Each two hexadecimal digits correspond to one byte of binary data, and each byte corresponds to one character.

** The character # represents a blank space.
**$CPTDWw. Informat**

Reads a character string that is encoded in Hebrew DOS (cp862) and then converts the character string to Windows (cp1255) encoding  

Category: Hebrew text handling  
Alignment: left  

**$CPTWDw. Informat**

Reads a character string that is encoded in Windows (cp1255) and then converts the character string to Hebrew DOS (cp862) encoding  

Category: Hebrew text handling  
Alignment: left  

**$EBCDICw. Informat**

Converts EBCDIC character data to native format  

Category: Character

**Syntax**  

$EBCDICw.$

**Syntax Description**

\[w\]

specifies the width of the input field.  

**Default:** 1 if the length of the variable is undefined; otherwise, the length of the variable  

**Range:** 1–32767

**Details**

If EBCDIC is the native format, no conversion occurs.

**Comparisons**

- On an IBM mainframe system, $EBCDICw.$ behaves like the $CHARw.$ informat.
On all other systems, $EBCDICw$. converts EBCDIC data to ASCII.

**Examples**

```plaintext
input @1 name $ebcdic3.
```

<table>
<thead>
<tr>
<th>When the Data Line = ...</th>
<th>ASCII</th>
<th>EBCDIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>qr`s</td>
<td>717273</td>
<td>9899A2</td>
</tr>
<tr>
<td>QRS</td>
<td>515253</td>
<td>D8D9E2</td>
</tr>
<tr>
<td>+;&gt;</td>
<td>2B3B3E</td>
<td>4E5E6E</td>
</tr>
</tbody>
</table>

* The results are hexadecimal representations of codes for characters. Each two hexadecimal digits correspond to one byte of binary data, and each byte corresponds to one character value.

---

**$HEXw. Informat**

Converts hexadecimal data to character data

**Category:** Character

**See:** $HEXw. Informat in the documentation for your operating environment.

---

**Syntax**

$HEXw.

**Syntax Description**

**w**

specifies the number of digits of hexadecimal data.

If $w=1$, $HEXw$. pads a trailing hexadecimal 0. If $w$ is an odd number that is greater than 1, then $HEXw$. reads $w-1$ hexadecimal characters.

**Default:** 2

**Range:** 1–32767

**Details**

The $HEXw$. informat converts every two digits of hexadecimal data into one byte of character data. Use $HEXw.$ to encode hexadecimal values into a character variable when your input method is limited to printable characters.

**Comparisons**

The HEX$w$. informat reads two digits of hexadecimal data at a time and converts them into one byte of numeric data.
Examples

    input @1 name $hex4.;

    When the data line = ...    The result is ...
    --------+--------+    ASCII  EBCDIC
    6C6C     11     %%

$KANJIm. Informat

Removes shift code data from DBCS data
Category:  DBCS

$KANJIXm. Informat

Adds shift code data to DBCS data
Category:  DBCS

$LOGVSm. Informat

Reads a character string that is in left-to-right logical order and then converts the character string to visual order
Category:  BIDI text handling
Alignment:  left

$LOGVSRm. Informat

Reads a character string that is in right-to-left logical order and then converts the character string to visual order
$OCTALw. Informat

Converts octal data to character data

Category: Character

Syntax
$OCTALw.

Syntax Description

w

specifies the width of the input field in bits. Because one digit of octal data represents three bits of binary information, increment the value of w by three for every column of octal data that $OCTALw. will read.

Default: 3
Range: 1–32767

Details

Eight bits of binary data represent the code for one digit of character data. Therefore, you need at least three digits of octal data to represent one digit of character data, which includes an extra bit. $OCTALw. treats every three digits of octal data as one digit of character data, ignoring the extra bit.

Use $OCTALw. to read octal representations of binary codes for unprintable characters. Enter an ASCII or EBCDIC equivalent for a particular character in octal notation. Then use $OCTALw. to convert it to its equivalent character value.

Use only the digits 0 through 7 in the input, with no embedded blanks. $OCTALw. ignores leading and trailing blanks.

Comparisons

The OCTALw. informat reads octal data and converts them into the numeric equivalents.

Examples

input @1 name $octal9.;
When the Data Line = ... The Result is ...

| ----+---- | EBCDIC | ASCII |
|------+------|--------+-------|
| 114  | <     | L      |

$\textit{PHEXw. Informat}$

Converts packed hexadecimal data to character data

**Category:** Character

**Syntax**

$\textit{PHEXw.}$

**Syntax Description**

$w$

specifies the number of bytes in the input.

When you use $\textit{PHEXw.}$ to read packed hexadecimal data, the length of the variable is the number of bytes that are required to store the resulting character value, not $w$. In general, a character variable whose length is implicitly defined with $\textit{PHEXw.}$ has a length of $2w-1$.

**Default:** 2

**Range:** 1–32767

**Details**

Packed hexadecimal data are like packed decimal data, except that all hexadecimal digits are valid. In packed hexadecimal data, the value of the low-order nibble has no meaning. In packed decimal data, the value of the low-order nibble indicates the sign of the numeric value that the data represent. The $\textit{PHEXw.}$ informat returns a character value and treats the value of the sign nibble as if it were $\text{x'F'}$, regardless of its actual value.

**Comparisons**

The PD$w.d.$ informat reads packed decimal data and converts them to numeric data.

**Examples**

input @1 devaddr $\text{phex2.};$
When the Data Line* = ... The Result is ...

| 0001111000001111 | 1E0 |

*The data line represents two bytes of actual binary data, with each half byte corresponding to a single hexadecimal digit. The equivalent hexadecimal representation for the data line is 1E0F.

$QUOTEw. Informat

Removes matching quotation marks from character data

Category: Character

Syntax

$QUOTEw.

Syntax Description

$w$

specifies the width of the input field.

Default: 8 if the length of the variable is undefined; otherwise, the length of the variable

Range: 1–32767

Examples

input @1 name $quote7.;

When the Data Line = ... The Result is ...

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>'SAS'</td>
<td>SAS</td>
</tr>
<tr>
<td>&quot;SAS&quot;</td>
<td>SAS</td>
</tr>
<tr>
<td>&quot;SAS's&quot;</td>
<td>SAS's</td>
</tr>
</tbody>
</table>
$REVERJw. Informat

Reads character data from right to left and preserves blanks

Category: Character

$REVERSw. Informat

Reads character data from right to left and left, and then left aligns the text

Category: Character

$UCS2Bw. Informat

Reads a character string that is encoded in big-endian, 16-bit, universal character set code in 2 octets (UCS2), Unicode encoding, and then converts the character string to the encoding of the current SAS session

Category: Character
Alignment: Left

$UCS2BEw. Informat

Reads a character string that is in the encoding of the current SAS session and then converts the character string to big-endian, 16-bit, universal character set code in 2 octets (UCS2), Unicode encoding

Category: Character

$UCS2Lw. Informat

Reads a character string that is encoded in little-endian, 16-bit, universal character set code in 2 octets (UCS2), Unicode encoding, and then converts the character string to the encoding of the current SAS session
$UCS2LEw. Informat

Reads a character string that is in the encoding of the current SAS session and then converts the character string to little-endian, 16-bit, universal character set code in 2 octets (UCS2), Unicode encoding

Category: Character

$UCS2Xw. Informat

Reads a character string that is encoded in 16-bit, universal character set code in 2 octets (UCS2), Unicode encoding, and then converts the character string to the encoding of the current SAS session

Category: Character
Alignment: Left

$UCS2XEw. Informat

Reads a character string that is in the encoding of the current SAS session and then converts the character string to 16-bit, universal character set code in 2 octets (UCS2), Unicode encoding

Category: Character

$UCS4Bw. Informat

Reads a character string that is encoded in big-endian, 32-bit, universal character set code in 4 octets (UCS4), Unicode encoding, and then converts the character string to the encoding of the current SAS session

Category: Character

$UCS4Lw.d Informat

Reads a character string that is encoded in little-endian, 32-bit, universal character set code in 4 octets (UCS4), Unicode encoding, and then converts the character string to the encoding of the current SAS session

Category: Character

$UCSXw. Informat

Reads a character string that is encoded in 32-bit, universal character set code in 4 octets (UCS4), Unicode encoding, and then converts the character string to the encoding of the current SAS session

Category: Character
Alignment: Left

$UCXSEw. Informat

Reads a character string that is in the encoding of the current SAS session and then converts the character string to 32-bit, universal character set code in 4 octets (UCS4), Unicode encoding

Category: Character

$UESCw. Informat

Reads a character string that is encoded in Unicode escape (UESC) representation, and then converts the character string to the encoding of the current SAS session

Category: Character
**$UESCEw. Informat**

Reads a character string that is in the encoding of the current SAS session, and then converts the character string to Unicode escape (UESC)

Category: Character


---

**$UNCRe. Informat**

Reads the numeric character representation (NCR) character string, and then converts the character string to the encoding of the current SAS session

Category: Character


---

**$UNCREw. Informat**

Reads a character string in the encoding of the current SAS session, and then converts the character string to session-encoded NCR (numeric character representation)

Category: Character


---

**$UPARENw. Informat**

Reads a character string that is encoded in UPAREN (Unicode parenthesis) representation, and then converts the character string to the encoding of the current SAS session

Category: Character


---

**$UPARENEw. Informat**

Reads a character string that is encoded in the current SAS session, and then converts the character string to the encoding of the Unicode parenthesis (UPAREN) representation
$UPARENPw. Informat

Reads a character string that is encoded in Unicode parenthesis (UPAREN) representation, and then converts the character string to the encoding of the current SAS session with national characters remaining in the encoding of the UPAREN representation.

Category: Character

$UPCASEw. Informat

Converts character data to uppercase

Category: Character

Syntax

$UPCASEw.

Syntax Description

`w` specifies the width of the input field.

Default: 8 if the length of the variable is undefined; otherwise, the length of the variable

Range: 1–32767

Details

Special characters, such as hyphens, are not altered.

Examples

```
input @1 name $upcase3.;
```

When the Data Line = ... The Result is ...


| -----+----- |
|-------+-------|
| `sas` | `SAS`  |
$\text{UTF8X}_w$. Informat

Reads a character string that is encoded in Unicode transformation format (UTF-8), and then converts the character string to the encoding of the current SAS session

Category: Character

Alignment: Left


$\text{VARYING}_w$. Informat

Reads character data of varying length

Valid: in a DATA step

Category: Character

Syntax

$\text{VARYING}_w$. length-variable

Syntax Description

\( w \)

specifies the maximum width of a character field for all the records in an input file.

Default: 8 if the length of the variable is undefined; otherwise, the length of the variable

Range: 1–32767

length-variable

specifies a numeric variable that contains the width of the character field in the current record. SAS obtains the value of length-variable by reading it directly from a field that is described in an INPUT statement or by calculating its value in the DATA step.

Requirement: You must specify length-variable immediately after $\text{VARYING}_w$. in an INPUT statement.

Restriction: Length-variable cannot be an array reference.

Tip: If the value of length-variable is 0, negative, or missing, SAS reads no data from the corresponding record. This enables you to read zero-length records and fields. If length-variable is greater than 0 but less than \( w \), SAS reads the number of columns that are specified by length-variable. Then SAS pads the value with
trailing blanks up to the maximum width that is assigned to the variable. If
`length-variable` is greater than or equal to `w`, SAS reads `w` columns.

Details

Use `$VARYINGw.` when the length of a character value differs from record to record. After reading a data value with `$VARYINGw`, the pointer's position is set to the first column after the value.

Examples

**Example 1: Obtaining a Current Record Length Directly**

```sas
input fwidth 1. name $varying9. fwidth;
```

<table>
<thead>
<tr>
<th>Data Line* = ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>5shark</td>
<td>shark</td>
</tr>
<tr>
<td>3sunfish</td>
<td>sun</td>
</tr>
<tr>
<td>8bluefish</td>
<td>bluefish</td>
</tr>
</tbody>
</table>

* Notice the result of reading the second data line.

**Example 2: Obtaining a Record Length Indirectly**

Use the `LENGTH=` option in the `INFILE` statement to obtain a record length indirectly. The input data lines and results follow the explanation of the SAS statements.

```sas
data one;
  infile file-specification length=reclen;
  input @;
  fwidth=reclen-9;
  input name $ 1-9
    @10 class $varying20. fwidth;
run;
```

The `LENGTH=` option in the `INFILE` statement assigns the internally stored record length to `RECLEN` when the first `INPUT` statement executes. The trailing `@` holds the record for another `INPUT` statement. Next, the assignment statement calculates the value of the varying-length field by subtracting the fixed-length portion of the record from the total record length. The variable `FWIDTH` contains the length of the last field and becomes the `length-variable` argument to the `$VARYING20.` informat.

<table>
<thead>
<tr>
<th>Data Lines</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>----+----1----+----2</td>
<td>----+----1----+----2</td>
</tr>
<tr>
<td>PATEL CHEMISTRY</td>
<td>PATEL CHEMISTRY</td>
</tr>
<tr>
<td>JOHNSON GEOLOGY</td>
<td>JOHNSON GEOLOGY</td>
</tr>
<tr>
<td>WILCOX ART</td>
<td>WILCOX ART</td>
</tr>
</tbody>
</table>
$VSLOGw. Informat

Reads a character string that is in visual order and then converts the character string to left-to-right logical order

Category: BIDI text handling

$VSLOGRw. Informat

Reads a character string that is in visual order and then converts the character string to right-to-left logical order

Category: BIDI text handling

$w. Informat

Reads standard character data

Category: Character
Alias: $Fw.

Syntax

$w.

Syntax Description

$w

specifies the width of the input field. You must specify $w$ because SAS does not supply a default value.

Range: 1–32767

Details

The $w$. informat trims leading blanks and left aligns the values before storing the text. In addition, if a field contains only blanks and a single period, $w$. converts the period to a blank because it interprets the period as a missing value. The $w$. informat treats two or more periods in a field as character data.
Comparisons

The $w$. informat is almost identical to the $CHARw$. informat. However, $CHARw$. does not trim leading blanks nor does it convert a single period in an input field to a blank, while $w$. does both.

Examples

```
input @1 name $5.;
```

<table>
<thead>
<tr>
<th>When the Data Line = ...</th>
<th>The Result* is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>XYZ XYZ##</td>
<td>XYZ##</td>
</tr>
<tr>
<td>XYZ XYZ##</td>
<td>XYZ##</td>
</tr>
<tr>
<td>.</td>
<td>XYZ#YZ#</td>
</tr>
</tbody>
</table>

* The character # represents a blank space.

ANYDTDTEw. Informat

Reads and extracts date values from DATE, DATETIME, DDMMYY, JULIAN, MMDDYY, MONYY, TIME, YYMMDD, or YYQ informat values

Category: Date and Time

Syntax

ANYDTDTEw.

Syntax Description

\(w\)

specifies the width of the input field.

Default: 9

Range: 5-32

Details

The ANYDTDTE informat reads input data that corresponds to any of the following informats. The ANYDTDTE informat extracts the date part from the derived value.

DATE
DATETIME
DDMMYY
JULIAN
MMDDYY
MONYY
TIME
YYMMDD
YYQ

If the input value is a time-only value, then SAS assumes a date of 01JAN1960. The input values for all of the above informats are mutually exclusive except for MMDDYY, DDMMYY, and YYMMDD when two-digit years are used. It is possible for input data such as 01–02–03 to be ambiguous with respect to the month, day, and year. In this case, the DATESTYLE system option indicates the order of the month, day, and year.

**Comparisons**

The ANYTDTE informat extracts the date part from the derived value. The ANYDTDTM informat extracts the datetime part. The ANYDTTME informat extracts the time part.

**Examples**

```plaintext
input dateinfo anytdte21.;
```

<table>
<thead>
<tr>
<th>Data Lines</th>
<th>Informat</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>14JAN1921</td>
<td>DATE</td>
<td>-14231</td>
</tr>
<tr>
<td>14JAN1921 12:24:32.8</td>
<td>DATETIME</td>
<td>-14231</td>
</tr>
<tr>
<td>14011921</td>
<td>DDMMYY</td>
<td>-14231</td>
</tr>
<tr>
<td>1921014</td>
<td>JULIAN</td>
<td>-14231</td>
</tr>
<tr>
<td>01141921</td>
<td>MMDDYY</td>
<td>-14231</td>
</tr>
<tr>
<td>JAN1921</td>
<td>MONYY</td>
<td>-14244</td>
</tr>
<tr>
<td>12:24:32.8</td>
<td>TIME</td>
<td>0</td>
</tr>
<tr>
<td>19210114</td>
<td>YYMMDD</td>
<td>-14231</td>
</tr>
<tr>
<td>21Q1</td>
<td>YYQ</td>
<td>-14244</td>
</tr>
</tbody>
</table>

**See Also**

Informats:

“ANYDTDTMw. Informat” on page 970
“ANYDTTMEw. Informat” on page 971
ANYDTDTMw. Informat

Reads and extracts datetime values from DATE, DATETIME, DDMMYY, JULIAN, MMDDYY, MONYY, TIME, YYMMDD, or YYQ informat values.

Category: Date and Time

Syntax

ANYDTDTMw.

Syntax Description

w

specifies the width of the input field.

Default: 19
Range: 1-32

Details

The ANYDTDTM informat reads input data that corresponds to any of the following informats. The ANYDTDTM informat extracts the datetime part from the derived value.

- DATE
- DATETIME
- DDMMYY
- JULIAN
- MMDDYY
- MONYY
- TIME
- YYMMDD
- YYQ

If the input value is a time-only value, then SAS assumes a date of 01JAN1960. If the input value is a date value only, then SAS assumes a time of 12:00 midnight.

The input values for all of the above informats are mutually exclusive except for MMDDYY, DDMMYY, or YYMMDD when two-digit years are used. It is possible for input data such as 01–02–03 to be ambiguous with respect to the month, day, and year. In this case, the DATESTYLE system option indicates the order of the month, day, and year.

Comparisons

The ANYDTDTE informat extracts the date part from the derived value. The ANYDTDTM informat extracts the datetime part. The ANYDTTME informat extracts the time part.

Examples
input dateinfo anydtutm21.;

<table>
<thead>
<tr>
<th>Data Lines</th>
<th>Informat</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>14JAN1921</td>
<td>DATE</td>
<td>-1229558400</td>
</tr>
<tr>
<td>14JAN1921 12:24:32.8</td>
<td>DATETIME</td>
<td>-1229513727</td>
</tr>
<tr>
<td>14011921</td>
<td>DDMMYY</td>
<td>-1229558400</td>
</tr>
<tr>
<td>19210101</td>
<td>JULIAN</td>
<td>-1229558400</td>
</tr>
<tr>
<td>01141921</td>
<td>MMDDYY</td>
<td>-1229558400</td>
</tr>
<tr>
<td>JAN1921</td>
<td>MONYY</td>
<td>-1230681600</td>
</tr>
<tr>
<td>12:24:32.8</td>
<td>TIME</td>
<td>44672</td>
</tr>
<tr>
<td>19210101</td>
<td>YYMMDD</td>
<td>-1229558400</td>
</tr>
<tr>
<td>21Q1</td>
<td>YYQ</td>
<td>-1230681600</td>
</tr>
</tbody>
</table>

See Also

Informats:

“ANYDTDTTEw. Informat” on page 968
“ANYDTTMEw. Informat” on page 971

ANYDTTMEw. Informat

Reads and extracts time values from DATE, DATETIME, DDMMYY, JULIAN, MMDDYY, MONYY, TIME, YYMMDD, or YYQ informat values

Category: Date and Time

Syntax

ANYDTTMEw.

Syntax Description

\( w \)

specifies the width of the input field.

Default: 8
Range: 1-32

Details

The ANYDTTME informat reads input data that corresponds to any of the following informats. The ANYDTTME informat extracts the time part from the derived value.
DATE
DATETIME
DDMMYY
JULIAN
MMDDYY
MONYY
TIME
YYMMDD
YYQ

If the input value is a time-only value, then SAS assumes a date of 01JAN1960. If
the input value is a date value only, then SAS assumes a time of 12:00 midnight.
The input values for all of the above informats are mutually exclusive except for
MMDDYY and DDMMYY when two-digit years are used. It is possible for input data
such as 01–02–03 to be ambiguous with respect to the month, day, and year. In this
case, the DATESTYLE system option indicates the order of the month, day, and year.

Comparisons
The ANYDTDTE informat extracts the date part from the derived value. The
ANYDTDTDM informat extracts the datetime part. The ANYDTTME informat extracts
the time part.

Examples

```plaintext
input dateinfo anydttme21.;
```

<table>
<thead>
<tr>
<th>Data Lines</th>
<th>Informat</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>14JAN1921</td>
<td>DATE</td>
<td>0.0</td>
</tr>
<tr>
<td>14JAN1921</td>
<td>DATETIME</td>
<td>44672.799999</td>
</tr>
<tr>
<td>14011921</td>
<td>DDMMYY</td>
<td>0.0</td>
</tr>
<tr>
<td>1921014</td>
<td>JULIAN</td>
<td>0.0</td>
</tr>
<tr>
<td>01141921</td>
<td>MMDDYY</td>
<td>0.0</td>
</tr>
<tr>
<td>JAN1921</td>
<td>MONYY</td>
<td>0.0</td>
</tr>
<tr>
<td>12:24:32.8</td>
<td>TIME</td>
<td>44672.8</td>
</tr>
<tr>
<td>19210114</td>
<td>YYMMDD</td>
<td>0.0</td>
</tr>
<tr>
<td>21Q1</td>
<td>YYQ</td>
<td>0.0</td>
</tr>
</tbody>
</table>

See Also

Informats:

“ANYDTDTEw. Informat” on page 968
BINARY\textit{w.d} Informat

Converts positive binary values to integers

Category: Numeric

Syntax

\texttt{BINARY\textit{w.d}}

Syntax Description

\textit{w}  
specifies the width of the input field.  
\textbf{Default:} 8  
\textbf{Range:} 1–64

\textit{d}  
optionally specifies the power of 10 by which to divide the value. SAS uses the \textit{d} value even if the data contain decimal points.  
\textbf{Range:} 0–31

Details

Use only the character digits 1 and 0 in the input, with no embedded blanks. BINARY\textit{w.d} ignores leading and trailing blanks.  
\textit{BINARY\textit{w.d}} cannot read negative values. It treats all input values as positive (unsigned).

Examples

\begin{verbatim}
input @1 value binary8.1;
\end{verbatim}

When the data line = ...  
The Result is ...

\begin{tabular}{l|l}
\hline  
----+-----1----+  
00001111 & 1.5  
\hline
\end{tabular}
**BITSw.d Informat**

Extracts bits

Category: Numeric

**Syntax**

BITSw.d

**Syntax Description**

\( w \)
specifies the number of bits to read.

Default: 1

Range: 1–64

\( d \)
specifies the zero-based offset.

Range: 0–63

**Details**

The BITSw.d informat extracts particular bits from an input stream and assigns the numeric equivalent of the extracted bit string to a variable. Together, the \( w \) and \( d \) values specify the location of the string you want to read.

This informat is useful for extracting data from system records that have many pieces of information packed into single bytes.

**Examples**

```plaintext
input @1 value bits4.1;
```

**Table 5.4**

<table>
<thead>
<tr>
<th>When the Data Line = ...</th>
<th>The Result* is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>8</td>
</tr>
</tbody>
</table>

*The EBCDIC binary code for a capital B is 11000010, and the ASCII binary code is 01000010.

The input pointer moves to column 2 \((d=1)\). Then the INPUT statement reads four bits \((w=4)\) which is the bit string 1000 and stores the numeric value 8, which is equivalent to this binary combination.
**BZw.d Informat**

Converts blanks to 0s

Category: Numeric

### Syntax

**BZw.d**

### Syntax Description

**w**
- specifies the width of the input field.
  - Default: 1
  - Range: 1–32

**d**
- optionally specifies the power of 10 by which to divide the value. If the data contain decimal points, the d value is ignored.
  - Range: 0–31

### Details

The **BZw.d** informat reads numeric values, converts any trailing or embedded blanks to 0s, and ignores leading blanks.

The **BZw.d** informat can read numeric values that are located anywhere in the field. Blanks can precede or follow the numeric value, and a minus sign must precede negative values. The **BZw.d** informat ignores blanks between a minus sign and a numeric value in an input field.

The **BZw.d** informat interprets a single period in a field as a 0. The informat interprets multiple periods or other nonnumeric characters in a field as a missing value.

To use **BZw.d** in a DATA step with list input, change the delimiter for list input with the DLM= option in the INFILE statement. By default, SAS interprets blanks between values in the data line as delimiters rather than 0s.

### Comparisons

The **BZw.d** informat converts trailing or embedded blanks to 0s. If you do not want to convert trailing blanks to 0s (for example, when reading values in E-notation), use either the **w.d** informat or the **Ew.d** informat instead.

### Examples

```plaintext
input @1 x bz4.;
```
When the Data Line = ... | The Result is ...

| ---+----+---- |
| 34 | 3400 |
| -2 | -200 |
| -2 | -201 |

**CBw.d Informat**

Reads standard numeric values from column-binary files

**Category:** Column Binary

**Syntax**

CBw.d

**Syntax Description**

\( w \)

specifies the width of the input field.

**Range:** 1–32

\( d \)

optionally specifies the power of 10 by which to divide the value. SAS uses the \( d \) value even if the data contain decimal points.

**Details**

The CBw.d informat reads standard numeric values from column-binary files and translates the data into standard binary format.

SAS first stores each column of column-binary data you read with CBw.d in two bytes and ignores the two high-order bits of each byte. If the punch codes are valid, SAS stores the equivalent numeric value into the variable that you specify. If the combinations are not valid, SAS assigns the variable a missing value and sets the automatic variable _ERROR_ to 1.

**Examples**

```plaintext
input @1 x cb8.;
```
When the Data Line* = ... The Result is ...

| 0009 | 9 |

* The data line is a hexadecimal representation of the column binary. The punch card column for the example data has row 9 punched. The binary representation is 0000 0000 0000 1001.

See Also

Informats:
“$CBw. Informat” on page 950
“PUNCH.d Informat” on page 1012
“ROWw.d Informat” on page 1018

COMMAw.d Informat

Removes embedded characters

Category: Numeric
Alias: DOLLARw.d

Syntax
COMMAw.d

Syntax Description

$w$
specifies the width of the input field.
Default: 1
Range: 1–32

$d$
optionally specifies the power of 10 by which to divide the value. If the data contain decimal points, the $d$ value is ignored.
Range: 0–31

Details

The COMMAw.d informat reads numeric values and removes embedded commas, blanks, dollar signs, percent signs, dashes, and right parentheses from the input data. The COMMAw.d informat converts a left parenthesis at the beginning of a field to a minus sign.
Comparisons

The COMMAXw.d informat operates like the COMMAXw.d informat, but it reverses the roles of the decimal point and the comma. This convention is common in European countries.

Examples

```
  input $x comma10.;
```

<table>
<thead>
<tr>
<th>When the Data Line</th>
<th>The Result is</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1,000,000</td>
<td>1000000</td>
</tr>
<tr>
<td>(500)</td>
<td>-500</td>
</tr>
</tbody>
</table>

COMMAXw.d Informat

Removes embedded characters

Category: Numeric
Alias: DOLLARXw.d

Syntax

COMMAXw.d

Syntax Description

```
  w
```

specifies the width of the input field.

Default: 1
Range: 1–32

```
  d
```

optionally specifies the power of 10 by which to divide the value. If the data contain a comma, which represents a decimal point, the d value is ignored.

Range: 0–31

Details

The COMMAXw.d informat reads numeric values and removes embedded periods, blanks, dollar signs, percent signs, dashes, and right parentheses from the input data.
The COMMAX\(w.d\) informat converts a left parenthesis at the beginning of a field to a minus sign.

**Comparisons**

The COMMAX\(w.d\) informat operates like the COMMA\(w.d\) informat, but it reverses the roles of the decimal point and the comma. This convention is common in European countries.

**Examples**

```
input @1 x commax10.;
```

<table>
<thead>
<tr>
<th>When the Data Line = ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1.000.000</td>
<td>1000000</td>
</tr>
<tr>
<td>(500)</td>
<td>-500</td>
</tr>
</tbody>
</table>

**DATE\(w\). Informat**

Reads date values in the form `ddmmmyy` or `ddmmmyyyy`.

**Category:** Date and Time

**Syntax**

```
DATE\(w\).
```

**Syntax Description**

\(w\)

specifies the width of the input field.

**Default:** 7

**Range:** 7–32

**Tip:** Use a width of 9 to read a 4-digit year.

**Details**

The date values must be in the form `ddmmmyy` or `ddmmmyyyy`, where

\(dd\)

is an integer from 01 through 31 that represents the day of the month.
**Chapter 5**

**Examples**

```plaintext
input calendar_date date11.;
```

<table>
<thead>
<tr>
<th>When the data line = ...</th>
<th>The result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>16mar99</td>
<td>14319</td>
</tr>
<tr>
<td>16 mar 99</td>
<td>14319</td>
</tr>
<tr>
<td>16-mar-1999</td>
<td>14319</td>
</tr>
</tbody>
</table>

**See Also**

- Format:  
  “DATEw. Format” on page 118
- Function:  
  “DATE Function” on page 478
- System Option:  
  “YEARCUTOFF= System Option” on page 1630

---

**DATETIMEw. Informat**

Reads datetime values in the form **ddmmyyy hh:mm:ss.ss or ddmmyyyyy hh:mm:ss.ss**

**Category**: Date and Time

**Syntax**

```
DATETIMEw.
```
Syntax Description

\( w \)

specifies the width of the input field.

Default: 18

Range: 13–40

Details

The datetime values must be in the following form: \( ddmmyy \) or \( ddmmyyyy \), followed by a blank or special character, followed by \( hh:mm:ss.ss \) (the time). In the date,

\( dd \)

is an integer from 01 through 31 that represents the day of the month.

\( mmm \)

is the first three letters of the month name.

\( yy \) or \( yyyy \)

is a two-digit or four-digit integer that represents the year.

In the time,

\( hh \)

is the number of hours ranging from 00 through 23.

\( mm \)

is the number of minutes ranging from 00 through 59.

\( ss.ss \)

is the number of seconds ranging from 00 through 59 with the fraction of a second following the decimal point.

\text{DATETIME}w.\text{ requires values for both the date and the time; however, the \( ss.ss \) portion is optional.}

\text{Note: SAS interprets a two-digit year as belonging to the 100-year span that is defined by the \text{YEARCUTOFF=} system option.}

\text{Note: SAS can read time values with AM and PM in them.}

Examples

```
input date_and_time datetime20.;
```

<table>
<thead>
<tr>
<th>When the Data Line</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>16mar97:11:23:07.4</td>
<td>1237202587.4</td>
</tr>
<tr>
<td>16mar1997/11:23:07.4</td>
<td>1237202587.4</td>
</tr>
<tr>
<td>16mar1997/11:23 PM</td>
<td>1237245780.0</td>
</tr>
</tbody>
</table>
See Also

Formats:
- “DATEw. Format” on page 118
- “DATETIMEw.d Format” on page 120
- “TIMEw.d Format” on page 209

Function:
- “DATETIME Function” on page 480

Informat:
- “DATEw. Informat” on page 979
- “TIMEw. Informat” on page 1037

System Option:
- “YEARCUTOFF= System Option” on page 1630

See the discussion on using SAS date and time values in SAS Language Reference: Concepts

**DDMMYYw. Informat**

Reads date values in the form `ddmmyy` or `ddmmyyyy`

Category: Date and Time

**Syntax**

`DDMMYYw.`

**Syntax Description**

`w`

specifies the width of the input field.

Default: 6

Range: 6–32

**Details**

The date values must be in the form `ddmmyy` or `ddmmyyyy`, where

`dd`

is an integer from 01 through 31 that represents the day of the month.

`mm`

is an integer from 01 through 12 that represents the month.

`yy` or `yyyy`

is a two-digit or four-digit integer that represents the year.

You can place blanks and other special characters between day, month, and year values. However, if you use delimiters, place them between all the values. Blanks can also be
placed before and after the date. Make sure the width of the input field allows space for blanks and special characters.

**Note:** SAS interprets a two-digit year as belonging to the 100-year span that is defined by the YEARCUTOFF= system option.

**Examples**

```plaintext
input calendar_date ddmmyy10.;
```

<table>
<thead>
<tr>
<th>When the Data Line = ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>160399</td>
<td>14319</td>
</tr>
<tr>
<td>16/03/99</td>
<td>14319</td>
</tr>
<tr>
<td>16 03 1999</td>
<td>14319</td>
</tr>
</tbody>
</table>

**See Also**

Formats:
- “DATEw. Format” on page 118
- “DDMMYYw. Format” on page 123
- “MMDDYYw. Format” on page 162
- “YYMMDDw. Format” on page 229

Function:
- “MDY Function” on page 643

Informats:
- “DATEw. Informat” on page 979
- “MMDDYYw. Informat” on page 993
- “YYMMDDw. Informat” on page 1044

System Option:
- “YEARCUTOFF= System Option” on page 1630

---

**Ew.d Informat**

Reads numeric values that are stored in scientific notation and double-precision scientific notation

Category: Numeric

See: Ew.d Informat in the documentation for your operating environment.
**Syntax**

\[ \text{E}w.d \]

**Syntax Description**

\( w \)

specifies the width of the field that contains the numeric value.

**Default:** 12

**Range:** 1–32

\( d \)

optionally specifies the number of digits to the right of the decimal point in the numeric value. If the data contain decimal points, the \( d \) value is ignored.

**Range:** 0–31

**Comparisons**

The \text{E}w.d\text{.} informat is not used extensively because the SAS informat for standard numeric data, the \text{w}d\text{.} informat, can read numbers in scientific notation. Use \text{E}w.d\text{.} to permit only scientific notation in your input data.

**Examples**

```
input @1 x e7.;
```

<table>
<thead>
<tr>
<th>When the Data Line = ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.257E3</td>
<td>1257</td>
</tr>
<tr>
<td>12d3</td>
<td>12000</td>
</tr>
</tbody>
</table>

**EURDFDEw. Informat**

Reads international date values

**Category:** Date and Time

**See:** The EURDFDE informat in \textit{SAS National Language Support (NLS): User's Guide}
**EURDFDT w. Informat**

Reads international datetime values in the form `ddmmmyy hh:mm:ss.ss` or `ddmmmyyyy hh:mm:ss.ss`

Category: Date and Time


---

**EURDFMY w. Informat**

Reads month and year date values in the form `mmmyy` or `mmmyyyy`

Category: Date and Time


---

**EURO w.d Informat**

Reads numeric values and removes embedded characters in European currency and reverses the comma and decimal point

Category: Numeric


---

**EUROX w.d Informat**

Reads numeric values and removes embedded characters in European currency

Category: Numeric


---

**FLOAT w.d Informat**

Reads a native single-precision, floating-point value and divides it by 10 raised to the $d$th power

Category: Numeric
Syntax

FLOAT\textit{w.d}

Syntax Description

\textit{w}

specifies the width of the input field.

\textbf{Requirement:} \textit{w} must be 4.

\textit{d}

optionally specifies the power of 10 by which to divide the value.

Details

The FLOAT\textit{w.d} informat is useful in operating environments where a float value is not the same as a truncated double.

On the IBM mainframe systems, a four-byte floating-point number is the same as a truncated eight-byte floating-point number. However, in operating environments that use the IEEE floating-point standard, such as the IBM PC-based operating environments and most UNIX platforms, a four-byte floating-point number is not the same as a truncated double. Therefore, the RB4. informat does not produce the same results as FLOAT4. Floating-point representations other than IEEE may have this same characteristic. Values read with FLOAT4. typically come from some other external program that is running in your operating environment.

Comparisons

The following table compares the names of float notation in several programming languages:

<table>
<thead>
<tr>
<th>Language</th>
<th>Float Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAS</td>
<td>FLOAT4.</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>REAL*4</td>
</tr>
<tr>
<td>C</td>
<td>float</td>
</tr>
<tr>
<td>IBM 370 ASM</td>
<td>E</td>
</tr>
<tr>
<td>PL/I</td>
<td>FLOAT BIN(21)</td>
</tr>
</tbody>
</table>

Examples

input x float4.;

<table>
<thead>
<tr>
<th>The Data Line* = ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{-----+----1-----+----2}</td>
<td></td>
</tr>
<tr>
<td>3F800000</td>
<td>1</td>
</tr>
</tbody>
</table>

* The data line is a hexadecimal representation of a binary number that is stored in IEEE form.
**HEXw. Informat**

Converts hexadecimal positive binary values to either integer (fixed-point) or real (floating-point) binary values.

**Category:** Numeric

**See:** HEXw. Informat in the documentation for your operating environment.

### Syntax

**HEXw.**

### Syntax Description

w

specifies the field width of the input value and also specifies whether the final value is fixed-point or floating-point.

**Default:** 8

**Range:** 1–16

**Tip:** If w<16, HEXw. converts the input value to positive integer binary values, treating all input values as positive (unsigned). If w is 16, HEXw. converts the input value to real binary (floating-point) values, including negative values.

### Details

**Note:** Different operating environments store floating-point values in different ways. However, HEX16. reads hexadecimal representations of floating-point values with consistent results if the values are expressed in the same way that your operating environment stores them. △

The HEXw. informat ignores leading or trailing blanks.

### Examples

```
input @1 x hex3. @5 y hex16.;
```

<table>
<thead>
<tr>
<th>When the Data Line* = ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>88F 4152000000000000 2191 5.125</td>
<td></td>
</tr>
</tbody>
</table>

* The data line shows IBM mainframe hexadecimal data.
Reads native integer binary (fixed-point) values, including negative values

Category: Numeric

See: IBw.d Informat in the documentation for your operating environment.

Syntax

IBw.d

Syntax Description

w
  specifies the width of the input field.
  Default: 4
  Range: 1–8

d
  optionally specifies the power of 10 by which to divide the value.
  Range: 0–10

Details

The IBw.d informat reads integer binary (fixed-point) values, including negative values represented in two’s complement notation. IBw.d reads integer binary values with consistent results if the values are created in the same type of operating environment that you use to run SAS.

  Note: Different operating environments store integer binary values in different ways. This concept is called byte ordering. For a detailed discussion about byte ordering, see “Byte Ordering for Integer Binary Data on Big Endian and Little Endian Platforms” on page 935.

Comparisons

The IBw.d and PIBw.d informats are used to read native format integers. (Native format allows you to read and write values created in the same operating environment.) The IBRw.d and PIBRw.d informats are used to read little endian integers in any operating environment.

To view a table that shows the type of informat to use with big endian and little endian integers, see Table 5.1 on page 936.

To view a table that compares integer binary notation in several programming languages, see Table 5.2 on page 936.

Examples

You can use the INPUT statement and specify the IB informat. However, these examples use the informat with the INPUT function, where binary input values are described using a hex literal.
x=input('0080',ib2.);
y=input('8000',ib2.);

When the SAS Statement is ...
The Result on Big Endian Platforms is ...
The Result on Little Endian Platforms is ...

| put x=;     | 128            | -32768        |
| put y=;     | -32768         | 128           |

See Also

Informat:  
“IBRw.d Informat” on page 989

**IBRw.d Informat**

Reads integer binary (fixed-point) values in Intel and DEC formats

**Category:** Numeric

**Syntax**

**IBRw.d**

**Syntax Description**

**w**

specifies the width of the input field.

Default: 4  
Range: 1–8

**d**

optionally specifies the power of 10 by which to divide the value.

Range: 0–10

**Details**

The IBRw.d informat reads integer binary (fixed-point) values, including negative values that are represented in two's complement notation. IBRw.d reads integer binary values that are generated by and for Intel and DEC platforms. Use IBRw.d to read integer binary data from Intel or DEC environments in other operating environments. The IBRw.d informat in SAS code allows for a portable implementation for reading the data in any operating environment.
Note: Different operating environments store integer binary values in different ways. This concept is called byte ordering. For a detailed discussion about byte ordering, see “Byte Ordering for Integer Binary Data on Big Endian and Little Endian Platforms” on page 935.

Comparisons

The IBw.d and PIBw.d informats are used to read native format integers. (Native format allows you to read and write values that are created in the same operating environment.) The IBRw.d and PIBRw.d informats are used to read little endian integers in any operating environment.

On Intel and DEC operating environments, the IBw.d and IBRw.d informats are equivalent.

To view a table that shows the type of informat to use with big endian and little endian integers, see Table 5.1 on page 936.

To view a table that compares integer binary notation in several programming languages, see Table 5.2 on page 936.

Examples

You can use the INPUT statement and specify the IBR informat. However, in these examples we use the informat with the INPUT function, where binary input values are described using a hex literal.

```sas
x=input('0100'x,ibr2.);
y=input('0001'x,ibr2.);
```

<table>
<thead>
<tr>
<th>When the SAS Statement is ...</th>
<th>The Result on Big Endian Platforms is ...</th>
<th>The Result on Little Endian Platforms is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>put x=;</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>put y=;</td>
<td>256</td>
<td>256</td>
</tr>
</tbody>
</table>

See Also

Informat:

“IBw.d Informat” on page 988

### IEEEw.d Informat

Reads an IEEE floating-point value and divides it by 10 raised to the $d$th power

**Category:** Numeric

**Syntax**

```sas
IEEEw.d
```
Syntax Description

\( w \)

specifies the width of the input field.

**Default:** 8

**Range:** 2–8

**Tip:** If \( w \) is 8, an IEEE double-precision, floating-point number is read. If \( w \) is 5, 6, or 7, an IEEE double-precision, floating-point number is read, which assumes truncation of the appropriate number of bytes. If \( w \) is 4, an IEEE single-precision, floating-point number is read. If \( w \) is 3, an IEEE single-precision, floating-point number is read, which assumes truncation of one byte.

\( d \)

specifies the power of 10 by which to divide the value.

Details

The IEEE\( w.d \) informat is useful in operating environments where IEEE is the floating-point representation that is used. In addition, you can use the IEEE\( w.d \) informat to read files that are created by programs on operating environments that use the IEEE floating-point representation.

Typically, programs generate IEEE values in single precision (4 bytes) or double precision (8 bytes). Truncation is performed by programs solely to save space on output files. Machine instructions require that the floating-point number be of one of the two lengths. The IEEE\( w.d \) informat allows other lengths, which enables you to read data from files that contain space-saving truncated data.

Examples

```plaintext
input test1 ieee4.;
input test2 ieee5.;
```

When the Data Line* is ... The Result is ...

<table>
<thead>
<tr>
<th>Data Line</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>3F800000</td>
<td>1</td>
</tr>
<tr>
<td>3FP000000</td>
<td>1</td>
</tr>
</tbody>
</table>

* The data lines are hexadecimal representations of binary numbers that are stored in IEEE format.

The first INPUT statement reads the first data line, and the second INPUT statement reads the next data line.

**JDATEYMD\( w \). Informat**

Reads Japanese Kanji date values in the format yymmdd or yyyyymmdd
JNENGOw. Informat

Reads Japanese Kanji date values in the form yymmd

Category: Date and Time
Alignment: left

JULIANw. Informat

Reads Julian dates in the form yydd or yyyydd

Category: Date and Time

Syntax

JULIANw.

Syntax Description

w

specifies the width of the input field.

Default: 5
Range: 5–32

Details

The date values must be in the form yydd or yyyydd, where

yy or yyyy

is a two-digit or four-digit integer that represents the year.

dd or ddd

is an integer from 01 through 365 that represents the day of the year.

Julian dates consist of strings of contiguous numbers, which means that zeros must pad any space between the year and the day values.

Julian dates that contain year values before 1582 are invalid for the conversion to Gregorian dates.

Note: SAS interprets a two-digit year as belonging to the 100-year span that is defined by the YEARCUTOFF= system option.
Examples

```plaintext
input julian_date julian7.;
```

When the Data Line = ... The Result* is ...

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>99075</td>
<td>14319</td>
</tr>
<tr>
<td>1999075</td>
<td>14319</td>
</tr>
</tbody>
</table>

* The input values correspond to the seventy-fifth day of 1999, which is March 16.

See Also

Format:
“JULIANw. Format” on page 161

Functions:
“DATEJUL Function” on page 479
“JULDATE Function” on page 613

System Option:
“YEARCUTOFF= System Option” on page 1630

MINGUOW. Informat

Reads dates in Taiwanese form

Category: Date and Time


MMDDYYw. Informat

Reads date values in the form mmdyy or mmddyyyy

Category: Date and Time

Syntax

MMDDYYw.
Syntax Description

\( w \)

specifies the width of the input field.

**Default:** 6  
**Range:** 6–32

**Details**

The date values must be in the form \( mmddyy \) or \( mmddyyyy \), where

\( mm \)

is an integer from 01 through 12 that represents the month.

\( dd \)

is an integer from 01 through 31 that represents the day of the month.

\( yy \) or \( yyyy \)

is a two-digit or four-digit integer that represents the year.

You can separate the month, day, and year fields by blanks or by special characters. However, if you use delimiters, place them between all fields in the value. Blanks can also be placed before and after the date.

**Note:** SAS interprets a two-digit year as belonging to the 100-year span that is defined by the YEARCUTOFF= system option.

**Examples**

```
input calendar_date mmddyy8.;
```

<table>
<thead>
<tr>
<th>When the Data Line = ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>031699</td>
<td>14319</td>
</tr>
<tr>
<td>03/16/99</td>
<td>14319</td>
</tr>
<tr>
<td>03 16 99</td>
<td>14319</td>
</tr>
<tr>
<td>03161999</td>
<td>14319</td>
</tr>
</tbody>
</table>

**See Also**

**Formats:**

- “DATEw. Format” on page 118  
- “DDMMYYw. Format” on page 123  
- “MMDDYYw. Format” on page 162  
- “YYMMDDw. Format” on page 229

**Functions:**
“DAY Function” on page 481
“MDY Function” on page 643
“MONTH Function” on page 655
“YEAR Function” on page 916

Informats:
“DATEw. Informat” on page 979
“DDMMYYw. Informat” on page 982
“YYMMDDw. Informat” on page 1044

System Option:
“YEARCUTOFF= System Option” on page 1630

MONYYw. Informat

Reads month and year date values in the form mmmyy or mmmyyyy

Category: Date and Time

Syntax
MONYYw.

Syntax Description

w
specifies the width of the input field.

Default: 5
Range: 5–32

Details

The date values must be in the form mmmyy or mmmyyyy, where

mmm
is the first three letters of the month name.

yy or yyyy
is a two-digit or four-digit integer that represents the year.

A value read with the MONYYw. informat results in a SAS date value that corresponds to the first day of the specified month.

Note: SAS interprets a two-digit year as belonging to the 100-year span that is defined by the YEARCUTOFF= system option.
Examples

input month_and_year monyy7.;

<table>
<thead>
<tr>
<th>When the Data Line = ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>mar 99</td>
<td>14304</td>
</tr>
<tr>
<td>mar1999</td>
<td>14304</td>
</tr>
</tbody>
</table>

See Also

Formats:
- “DDMMYYw. Format” on page 123
- “MMDDYYw. Format” on page 162
- “MONYYw. Format” on page 172
- “YYMMDDw. Format” on page 229

Functions:
- “MONTH Function” on page 655
- “YEAR Function” on page 916

Informat:
- “DDMMYYw. Informat” on page 982
- “MMDDYYw. Informat” on page 993
- “YYMMDDw. Informat” on page 1044

System Option:
- “YEARCUTOFF= System Option” on page 1630

MSECw. Informat

Reads TIME MIC values

Category: Date and Time

Syntax

MSECw.
**Syntax Description**

\( w \)

specifies the width of the input field.

**Requirement:** \( w \) must be 8 because the OS TIME macro or the STCK System/370 instruction on IBM mainframes each return an eight-byte value.

**Details**

The MSEC\( w \). informat reads time values that are produced by IBM mainframe operating environments and converts the time values to SAS time values.

Use the MSEC\( w \). informat to find the difference between two IBM mainframe TIME values, with precision to the nearest microsecond.

**Comparisons**

The MSEC\( w \). and TODSTAMP\( w \). informats both read IBM time-of-day clock values, but the MSEC\( w \). informat assigns a time value to a variable, and the TODSTAMP\( w \). informat assigns a datetime value.

**Examples**

```sas
input btime msec8.;
```

<table>
<thead>
<tr>
<th>Data Line* = ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000EA044E65A000</td>
<td>62818.412122</td>
</tr>
</tbody>
</table>

* The data line is a hexadecimal representation of a binary 8-byte time-of-day clock value. Each byte occupies one column of the input field. The result is a SAS time value corresponding to 5:26:58.41 PM.

**See Also**

Informat:

“TODSTAMP\( w \). Informat” on page 1039

---

**NENGO\( w \). Informat**

Reads Japanese date values in the form eeymmdd

**Category:** Date and Time

**See:** The NENGO informat in SAS National Language Support (NLS): User’s Guide
**NLDATEm. Informat**

Reads the date value in the specified locale and then converts the date value to the local SAS date value

Category: Date and Time


---

**NLDATMw. Informat**

Reads the datetime value of the specified locale, and then converts the datetime value to the local SAS datetime value

Category: Date and Time


---

**NLMNYw.d Informat**

Reads monetary data in the specified locale for the local expression, and converts the data to a numeric value

Category: Numeric


---

**NLMNYIw.d Informat**

Reads monetary data in the specified locale for the international expression, and then converts the data to a numeric value

Category: Numeric


---

**NLNUMw.d Informat**

Reads numeric data in the specified locale for local expressions, and then converts the data to a numeric value
NLNUMIw.d Informat

Reads numeric data in the specified locale for international expressions, and then converts the data to a numeric value

Category: Numeric


NLPCTw.d Informat

Reads percentage data in the specified locale for local expressions, and then converts the data to a numeric value

Category: Numeric


NLPCTIw.d Informat

Reads percentage data in the specified locale for international expressions, and then converts the data to a numeric value

Category: Numeric


NLTIMAPw. Informat

Reads the time value and uses a.m. and p.m. in the specified locale, and then converts the time value to the local SAS-time value

Category: Date and Time

**NLTIMEw. Informat**

 Reads the time value in the specified locale and then converts the time value to the local SAS time value  
 **Category:** Date and Time  
 **See:** NLTIMEw. Informat in *SAS National Language Support (NLS): User’s Guide*

**NUMXw.d Informat**

 Reads numeric values with a comma in place of the decimal point  
 **Category:** Numeric

**Syntax**

NUMX\textit{w.d}

**Syntax Description**

\textit{w}

 specifies the width of the input field.  
 **Default:** 12  
 **Range:** 1–32

\textit{d}

 optionally specifies the number of digits to the right of the decimal. If the data contain decimal points, the \textit{d} value is ignored.  
 **Range:** 0–31

**Details**

The NUMX\textit{w.d} informat reads numeric values and interprets a comma as a decimal point.

**Comparisons**

The NUMX\textit{w.d} informat is similar to the \textit{w.d} informat except that it reads numeric values that contain a comma in place of the decimal point.

**Examples**

```
  input @1 x numx10.;
```
OCTAL \( w.d \) Informat

Converts positive octal values to integers

Category: Numeric

Syntax

\texttt{OCTAL} \( w.d \)

Syntax Description

\( w \)

specifies the width of the input field.

\textbf{Default:} 3

\textbf{Range:} 1–24

\( d \)

optionally specifies the power of 10 by which to divide the value.

\textbf{Range:} 1–31

\textbf{Restriction:} must be greater than or equal to the \( w \) value.

Details

Use only the digits 0 through 7 in the input, with no embedded blanks. The OCTAL\( w.d \) informat ignores leading and trailing blanks. OCTAL\( w.d \) cannot read negative values. It treats all input values as positive (unsigned).

### See Also

Formats:

- “NUMX_{w.d} Format” on page 178
- “\( w.d \) Format” on page 214

<table>
<thead>
<tr>
<th>When the Data Line = ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>----+----1----+</td>
<td>896.48</td>
</tr>
<tr>
<td>896.48</td>
<td>896.48</td>
</tr>
<tr>
<td>3064.1</td>
<td>3064.1</td>
</tr>
<tr>
<td>6489</td>
<td>6489</td>
</tr>
</tbody>
</table>
Examples

    input @1 value octal3.1;

<table>
<thead>
<tr>
<th>Data Lines</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>177</td>
<td>12.7</td>
</tr>
</tbody>
</table>

**PDw.d Informat**

Reads data that are stored in IBM packed decimal format

**Category:** Numeric

**See:** PBw.d Informat in the documentation for your operating environment.

**Syntax**

`PDw.d`

**Syntax Description**

\[ w \]

specifies the width of the input field.

**Default:** 1

**Range:** 1–16

\[ d \]

optionally specifies the power of 10 by which to divide the value.

**Range:** 0–10

**Details**

The PDw.d informat is useful because many programs write data in packed decimal format for storage efficiency, fitting two digits into each byte and using only a half byte for a sign.

**Note:** Different operating environments store packed decimal values in different ways. However, PDw.d reads packed decimal values with consistent results if the values are created on the same type of operating environment that you use to run SAS.

**Comparisons**

The following table compares packed decimal notation in several programming languages:
Informats /52

<table>
<thead>
<tr>
<th>Language</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAS</td>
<td>PD4.</td>
</tr>
<tr>
<td>COBOL</td>
<td>COMP-3 PIC S9(7)</td>
</tr>
<tr>
<td>IBM 370 Assembler</td>
<td>PL4</td>
</tr>
<tr>
<td>PL/I</td>
<td>FIXED DEC</td>
</tr>
</tbody>
</table>

**Examples**

**Example 1: Reading Packed Decimal Data**

```
input @1 x pd4.;
```

<table>
<thead>
<tr>
<th>When the Data Line* = ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000128C</td>
<td>128</td>
</tr>
</tbody>
</table>

* The data line is a hexadecimal representation of a binary number stored in packed decimal form. Each byte occupies one column of the input field.

**Example 2: Creating a SAS Date with Packed Decimal Data**

```
input mnth pd4.;
date=input(put(mnth,6.),mmddyy6.);
```

<table>
<thead>
<tr>
<th>Data Lines*</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>0122599C</td>
<td>14603</td>
</tr>
</tbody>
</table>

* The data line is a hexadecimal representation of a binary number that is stored in packed decimal form on an IBM mainframe operating environment. Each byte occupies one column of the input field. The result is a SAS date value that corresponds to December 25, 1999.

**PDJULGw. Informat**

Reads packed Julian date values in the hexadecimal form yyyydddF for IBM

**Category:** Date and Time

**Syntax**

PDJULGw.
Syntax Description

\( w \)

specifies the width of the input field.

**Default:** 4  
**Range:** 4

Details

The PDJULG\( w \). informat reads IBM packed Julian date values in the form of \( yyyydddf \), converting them to SAS date values, where

\( yyyy \)

is the two-byte representation of the four-digit Gregorian year.

\( ddd \)

is the one-and-a-half byte representation of the three-digit integer that corresponds to the Julian day of the year, 1–365 (or 1–366 for leap years).

\( F \)

is the half byte that contains all binary 1s, which assigns the value as positive.

*Note:* SAS interprets a two-digit year as belonging to the 100-year span that is defined by the YEARCUTOFF= system option.

Examples

```
input date pdjulg4.;
```

<table>
<thead>
<tr>
<th>When Data Line = ...</th>
<th>The Result* is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1999003F</td>
<td>14247</td>
</tr>
</tbody>
</table>

* SAS date value 14247 represents January 3, 1999.

See Also

Formats:

“JULDAY\( w \). Format” on page 160  
“JULIAN\( w \). Format” on page 161  
“PDJULG\( w \). Format” on page 181  
“PDJULI\( w \). Format” on page 182

Functions:

“DATEJUL Function” on page 479  
“JULDATE Function” on page 613
Informats:

“JULIANw. Informat” on page 992
“PDJULIw. Informat” on page 1005

System Option:

“YEARCUTOFF= System Option” on page 1630

**PDJULIw. Informat**

Reads packed Julian dates in the hexadecimal format `ccyydddF` for IBM

Category: Date and Time

**Syntax**

PDJULIw.

**Syntax Description**

`w` specifies the width of the input field.

*Default:* 4

*Range:* 4

**Details**

The PDJULIw. informat reads IBM packed Julian date values in the form `ccyydddF`, converting them to SAS date values, where

`cc` is the one-byte representation of a two-digit integer that represents the century.

`yy` is the one-byte representation of a two-digit integer that represents the year. The PDJULIw informat makes an adjustment to the one-byte century representation by adding 1900 to the two-byte `ccyy` value in order to produce the correct four-digit Gregorian year. This adjustment causes `ccyy` values of 0098 to become 1998, 0101 to become 2001, and 0218 to become 2118.

`ddd` is the one-and-a-half bytes representation of the three-digit integer that corresponds to the Julian day of the year, 1–365 (or 1–366 for leap years).

`F` is the half byte that contains all binary 1s, which assigns the value as positive.
Examples

```
input date pdjuli4.;
```

<table>
<thead>
<tr>
<th>When the Data Line = ...</th>
<th>The Result* is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0099001F</td>
<td>14245</td>
</tr>
<tr>
<td>0110015F</td>
<td>18277</td>
</tr>
</tbody>
</table>

* SAS date value 14245 is January 1, 1999. SAS date value 18277 is January 15, 2010.

See Also

Formats:
- “JULDAYw. Format” on page 160
- “JULIANw. Format” on page 161
- “PDJULGw. Format” on page 181
- “PDJULIw. Format” on page 182

Functions:
- “DATEJUL Function” on page 479
- “JULDATE Function” on page 613

Informat:
- “JULIANw. Informat” on page 992
- “PDJULGw. Informat” on page 1003

System Option:
- “YEARCUTOFF= System Option” on page 1630

PDTIMEw. Informat

Reads packed decimal time of SMF and RMF records

Category: Date and Time

Syntax

```
PDTIMEw.
```
Syntax Description

\( w \)

specifies the width of the input field.

**Requirement:** \( w \) must be 4 because packed decimal time values in RMF and SMF records contain four bytes of information.

Details

The PDTIME\( w \). informat reads packed decimal time values that are contained in SMF and RMF records that are produced by IBM mainframe systems and converts the values to SAS time values.

The general form of a packed decimal time value in hexadecimal notation is \( 0hhmmssF \), where

- \( 0 \) is a half byte that contains all 0s.
- \( hh \) is one byte that represents two digits that correspond to hours.
- \( mm \) is one byte that represents two digits that correspond to minutes.
- \( ss \) is one byte that represents two digits that correspond to seconds.
- \( F \) is a half byte that contains all 1s.

If a field contains all 0s, PDTIME\( w \). treats it as a missing value.

PDTIME\( w \). enables you to read packed decimal time values from files that are created on an IBM mainframe on any operating environment.

Examples

```
input begin pdtime4.;
```

<table>
<thead>
<tr>
<th>Data Line</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0142225F</td>
<td>51745</td>
</tr>
</tbody>
</table>

The data line is a hexadecimal representation of a binary time value that is stored in packed decimal form. Each byte occupies one column of the input field. The result is a SAS time value that corresponds to 2:22.25 PM.

**PERCENT\( w \).d Informat**

Reads percentages as numeric values

**Category:** Numeric
Syntax

PERCENT\(w.d\)

Syntax Description

\(w\)
specifies the width of the input field.
- Default: 6
- Range: 1–32

\(d\)
optionally specifies the power of 10 by which to divide the value. If the data contain decimal points, the \(d\) value is ignored.
- Range: 0–31

Details

The PERCENT\(w.d\) informat converts the numeric portion of the input data to a number using the same method as the COMMA\(w.d\) informat. If a percent sign (%) follows the number in the input field, PERCENT\(w.d\) divides the number by 100.

Examples

```plaintext
input @1 x percent3. @4 y percent5.;
```

<table>
<thead>
<tr>
<th>When the Data Line = ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>---+-------+---+</td>
<td></td>
</tr>
<tr>
<td>1% (20%)</td>
<td>0.01 -0.2</td>
</tr>
</tbody>
</table>

PIB\(w.d\) Informat

Reads positive integer binary (fixed-point) values

Category: Numeric

See: PIB\(w.d\) Informat in the documentation for your operating environment.

Syntax

PIB\(w.d\)
**Syntax Description**

\( w \)

specifies the width of the input field.

- **Default:** 1
- **Range:** 1–8

\( d \)

optionally specifies the power of 10 by which to divide the value.

- **Range:** 0–10

**Details**

All values are treated as positive. PIB\( w \).\( d \) reads positive integer binary values with consistent results if the values are created in the same type of operating environment that you use to run SAS.

*Note:* Different operating environments store positive integer binary values in different ways. This concept is called byte ordering. For a detailed discussion about byte ordering, see “Byte Ordering for Integer Binary Data on Big Endian and Little Endian Platforms” on page 935.

**Comparisons**

- Positive integer binary values are the same as integer binary values except that the sign bit is part of the value, which is always a positive integer. The PIB\( w \).\( d \) informat treats all values as positive and includes the sign bit as part of the value.

- The PIB\( w \).\( d \) informat with a width of 1 results in a value that corresponds to the binary equivalent of the contents of a byte. This is useful if your data contain values between hexadecimal 80 and hexadecimal FF, where the high-order bit can be misinterpreted as a negative sign.

- The IB\( w \).\( d \) and PIB\( w \).\( d \) informats are used to read native format integers. (Native format allows you to read and write values that are created in the same operating environment.) The IBR\( w \).\( d \) and PIBR\( w \).\( d \) informats are used to read little endian integers in any operating environment.

  To view a table that shows the type of informat to use with big endian and little endian integers, see Table 5.1 on page 936.

  To view a table that compares integer binary notation in several programming languages, see Table 5.2 on page 936.

**Examples**

You can use the INPUT statement and specify the PIB informat. However, in these examples we use the informat with the INPUT function, where binary input values are described by using a hex literal.

```sas
x=input('0100'x,pib2.);
y=input('0001'x,pib2.);
```
When the SAS Statement is

<table>
<thead>
<tr>
<th></th>
<th>The Result on Big Endian Platforms is</th>
<th>The Result on Little Endian Platforms is</th>
</tr>
</thead>
<tbody>
<tr>
<td>put x=;</td>
<td>256</td>
<td>1</td>
</tr>
<tr>
<td>put y=;</td>
<td>1</td>
<td>256</td>
</tr>
</tbody>
</table>

See Also

Informat:

“PIBRw.d Informat” on page 1010

PIBRw.d Informat

Reads positive integer binary (fixed-point) values in Intel and DEC formats

Category: Numeric

Syntax

PIBRw.d

Syntax Description

\( w \)

specifies the width of the input field.

Default: 1

Range: 1–8

\( d \)

optionally specifies the power of 10 by which to divide the value.

Range: 0–10

Details

All values are treated as positive. PIBRw.d reads positive integer binary values that have been generated by and for Intel and DEC operating environments. Use PIBRw.d to read positive integer binary data from Intel or DEC environments on other operating environments. The PIBRw.d informat in SAS code allows for a portable implementation for reading the data in any operating environment.

Note: Different operating environments store positive integer binary values in different ways. This concept is called byte ordering. For a detailed discussion about byte ordering, see “Byte Ordering for Integer Binary Data on Big Endian and Little Endian Platforms” on page 935.
Comparisons

- Positive integer binary values are the same as integer binary values except that the sign bit is part of the value, which is always a positive integer. The PIBRw.d informat treats all values as positive and includes the sign bit as part of the value.
- The PIBRw.d informat with a width of 1 results in a value that corresponds to the binary equivalent of the contents of a byte. This is useful if your data contain values between hexadecimal 80 and hexadecimal FF, where the high-order bit can be misinterpreted as a negative sign.
- On Intel and DEC platforms, the PIBw.d and PIBRw.d informats are equivalent.
- The IBw.d and PIBw.d informats are used to read native format integers. (Native format allows you to read and write values that are created in the same operating environment.) The IBRw.d and PIBRw.d informats are used to read little endian integers in any operating environment.

To view a table that shows the type of informat to use with big endian and little endian integers, see Table 5.1 on page 936.

To view a table that compares integer binary notation in several programming languages, see Table 5.2 on page 936.

Examples

You can use the INPUT statement and specify the PIBR informat. However, these examples use the informat with the INPUT function, where binary input values are described using a hex literal.

```sas
x=input('0100'x,pibr2.);
y=input('0001'x,pibr2.);
```

<table>
<thead>
<tr>
<th>When the SAS Statement is</th>
<th>The Result on Big Endian Platforms is</th>
<th>The Result on Little Endian Platforms is</th>
</tr>
</thead>
<tbody>
<tr>
<td>put x=;</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>put y=;</td>
<td>256</td>
<td>256</td>
</tr>
</tbody>
</table>

See Also

Informat:
“PIBw.d Informat” on page 1008

PKw.d Informat

Reads unsigned packed decimal data

Category: Numeric

Syntax

PKw.d
Syntax Description

\( w \)

specifies the number of bytes of unsigned packed decimal data, each of which contains two digits.

**Default:** 1

**Range:** 1–16

\( d \)

optionally specifies the power of 10 by which to divide the value.

**Range:** 0–10

**Details**

Each byte of unsigned packed decimal data contains two digits.

**Comparisons**

The PK\( w.d \) informat is the same as the PD\( w.d \) informat, except that PK\( w.d \) treats the sign half of the field's last byte as part of the value, not as the sign of the value.

**Examples**

```
input @1 x pk3.;
```

<table>
<thead>
<tr>
<th>When the Data Line* = ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>001234</td>
<td>1234</td>
</tr>
</tbody>
</table>

* The data line is a hexadecimal representation of a binary number stored in unsigned packed decimal form. Each byte occupies one column of the input field.

---

**PUNCH.d Informat**

**Reads whether a row of column-binary data is punched**

**Category:** Column Binary

**Syntax**

PUNCH.d
**Syntax Description**

\(d\)

specifies which row in a card column to read.

**Range:** 1–12

**Details**

This informat assigns the value 1 to the variable if row \(d\) of the current card column is punched, or 0 if row \(d\) of the current card column is not punched. After PUNCH \(d\) reads a field, the pointer does not advance to the next column.

**Examples**

<table>
<thead>
<tr>
<th>When the Data Line*</th>
<th>And the SAS Statement is ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>12-7-8</td>
<td>input x punch.12</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>input x punch.11</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>input x punch0.7</td>
<td>1</td>
</tr>
</tbody>
</table>

* The data line is punched card code. The punch card column for the example data has row 12, row 7, and row 8 punched.

**See Also**

Informats:

“$CBw. Informat” on page 950
“CBw.d Informat” on page 976
“ROWw.d Informat” on page 1018

**RBw.d Informat**

Reads numeric data that are stored in real binary (floating-point) notation

**Category:** Numeric

**See:** RBw.d Informat in the documentation for your operating environment.

**Syntax**

RBw.d
Syntax Description

\( w \)

specifies the width of the input field.

**Default:** 4

**Range:** 2–8

\( d \)

optionally specifies the power of 10 by which to divide the value.

**Range:** 0–10

Details

*Note:* Different operating environments store real binary values in different ways. However, the RB\(w.d\) informat reads real binary values with consistent results if the values are created on the same type of operating environment that you use to run SAS.

Comparisons

The following table compares the names of real binary notation in several programming languages:

<table>
<thead>
<tr>
<th>Language</th>
<th>4 Bytes</th>
<th>8 Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAS</td>
<td>RB4.</td>
<td>RB8.</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>REAL*4</td>
<td>REAL*8</td>
</tr>
<tr>
<td>C</td>
<td>float</td>
<td>double</td>
</tr>
<tr>
<td>IBM 370 assembler</td>
<td>F</td>
<td>D</td>
</tr>
<tr>
<td>PL/I</td>
<td>FLOAT BIN(21)</td>
<td>FLOAT BIN(53)</td>
</tr>
</tbody>
</table>

**CAUTION:**

Using the RB\(w.d\) informat to read real binary information on equipment that conforms to the IEEE standard for floating-point numbers results in a truncated eight-byte number (double-precision), rather than in a true four-byte floating-point number (single-precision).

Examples

```
input @1 x rb8.;
```

<table>
<thead>
<tr>
<th>When the Data Line* = ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>4280000000000000000000000</td>
<td>128</td>
</tr>
</tbody>
</table>

* The data line is a hexadecimal representation of a real binary (floating-point) number on an IBM mainframe operating environment. Each byte occupies one column of the input field.
See Also

Informat:
“IEEEw.d Informat” on page 990

RMFDURw. Informat

Reads duration intervals of RMF records
Category: Date and Time

Syntax
RMFDURw.

Syntax Description

\( w \)
specifies the width of the input field.

Requirement: \( w \) must be 4 because packed decimal duration values in RMF records contain four bytes of information.

Details
The RMFDURw. informat reads the duration of RMF measurement intervals of RMF records that are produced as packed decimal data by IBM mainframe systems and converts them to SAS time values.

The general form of the duration interval data in an RMF record in hexadecimal notation is \( mmsstttF \), where

\( mm \)
is the one-byte representation of two digits that correspond to minutes.

\( ss \)
is the one-byte representation of two digits that correspond to seconds.

\( ttt \)
is the one-and-a-half-bytes representation of three digits that correspond to thousandths of a second.

\( F \)
is a half byte that contains all binary 1s, which assigns the value as positive.

If the field does not contain packed decimal data, RMFDURw. results in a missing value.

Comparisons

- Both the RMFDURw. informat and the RMFSTAMPw. informat read packed decimal information from RMF records that are produced by IBM mainframe systems.
The RMFDUR\(_w\). informat reads duration data and results in a time value.

The RMFSTAMP\(_w\). informat reads time-of-day data and results in a datetime value.

**Examples**

```
input dura rmfdur4.;
```

<table>
<thead>
<tr>
<th>When the data line* = ...</th>
<th>The result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>3552226F</td>
<td>2152.226</td>
</tr>
</tbody>
</table>

* The data line is a hexadecimal representation of a binary duration value that is stored in packed decimal form as it would appear in an RMF record. Each byte occupies one column of the input field. The result is a SAS time value corresponding to 00:35:52.226.

**See Also**

Informats:

"RMFSTAMP\(_w\). Informat" on page 1016

"SMFSTAMP\(_w\). Informat" on page 1035
Details

The RMFSTAMPw. informat reads packed decimal time and date values of RMF records that are produced by IBM mainframe systems, and converts the time and date values to SAS datetime values.

The general form of the time and date information in an RMF record in hexadecimal notation is 0hhmmssFccyydddF, where

0  
is the half byte that contains all binary 0s.

hh  
is the one-byte representation of two digits that correspond to the hour of the day.

mm  
is the one-byte representation of two digits that correspond to minutes.

ss  
is 1 byte that represents two digits that correspond to seconds.

cc  
is the one-byte representation of two digits that correspond to the century.

yy  
is the one-byte representation of two digits that correspond to the year.

ddd  
is the one-and-a-half bytes that contain three digits that correspond to the day of the year.

F  
is the half byte that contains all binary 1s.

The century indicators 00 correspond to 1900, 01 to 2000, and 02 to 2100.

RMFSTAMPw. enables you to read, on any operating environment, packed decimal time and date values from files that are created on an IBM mainframe.

Comparisons

Both the RMFSTAMPw. informat and the PDTIMEw. informat read packed decimal values from RMF records. The RMFSTAMPw. informat reads both time and date values and results in a SAS datetime value. The PDTIMEw. informat reads only time values and results in a SAS time value.

Examples

```
input begin rmfstamp8.;
```

<table>
<thead>
<tr>
<th>When the Data Line* = ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0142225F0102286F</td>
<td>1350138145</td>
</tr>
</tbody>
</table>

* The data line is a hexadecimal representation of a binary time and date value that is stored in packed decimal form as it would appear in an RMF record. Each byte occupies one column
of the input field. The result is a SAS datetime value that corresponds to October 13, 2002, 2:22.25 PM.

---

**ROWw.d Informat**

**Reads a column-binary field down a card column**

**Category:** Column Binary

**Syntax**

\[ \text{ROW}w.d \]

**Syntax Description**

\[ w \]

specifies the row where the field begins.

**Range:** 0–12

\[ d \]

specifies the length in rows of the field.

**Default:** 1

**Range:** 1–25

**Details**

The ROWw.d informat assigns the relative position of the punch in the field to a numeric variable.

If the field that you specify has more than one punch, ROWw.d assigns the variable a missing value and sets the automatic variable _ERROR_ to 1. If the field has no punches, ROWw.d assigns the variable a missing value.

ROWw.d can read fields across columns, continuing with row 12 of the new column and going down through the rest of the rows. After ROWw.d reads a field, the pointer moves to the next row.

**Examples**

\[
\begin{align*}
\text{input } x & \text{ row5.3} \\
\text{input } x & \text{ row7.1} \\
\text{input } x & \text{ row5.2} \\
\text{input } x & \text{ row3.5}
\end{align*}
\]
When the Data Line* = ...

<table>
<thead>
<tr>
<th>Data Line</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>04 04 3</td>
<td>1 . 5</td>
</tr>
</tbody>
</table>

* The data line is a hexadecimal representation of the column binary. The punch card column for the example data has row 7 punched. The binary representation is 0000 0000 0000 0100.

**See Also**

Informs:
- “$CBw. Informat” on page 950
- “CBw.d Informat” on page 976
- “PUNCH.d Informat” on page 1012

---

**S370FFw.d Informat**

Reads EBCDIC numeric data

**Category:** Numeric

**Syntax**

S370FFw.d

**Syntax Description**

**w**

specifies the width of the input field.

- Default: 12
- Range: 1–32

**d**

optionally specifies the power of 10 by which to divide the value.

- Range: 0–31

**Details**

The S370FFw.d informat reads numeric data that are represented in EBCDIC and converts the data to native format. If EBCDIC is the native format, S370FFw.d performs no conversion.
S370FF\(w.d\) reads EBCDIC numeric values that are represented with one byte per digit. Use S370FF\(w.d\) on other operating environments to read numeric data from IBM mainframe files.

S370FF\(w.d\) reads numeric values located anywhere in the input field. EBCDIC blanks can precede or follow a numeric value with no effect. If a value is negative, an EBCDIC minus sign should immediately precede the value. S370FF\(w.d\) reads values with EBCDIC decimal points and values in scientific notation, and it interprets a single EBCDIC period as a missing value.

**Comparisons**

The S370FF\(w.d\) informat performs the same role for numeric data that the $EBCDIC\(w.d\)$ informat does for character data. That is, on an IBM mainframe system, S370FF\(w.d\) has the same effect as the standard \(w.d\) informat. On all other systems, using S370FF\(w.d\) is equivalent to using $EBCDIC\(w.d\)$ as well as using the standard \(w.d\) informat.

**Examples**

```plaintext
input @1 x s370ff3.;
```

<table>
<thead>
<tr>
<th>When the Data Line* is</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>(F1F2F3)</td>
<td>123</td>
</tr>
<tr>
<td>(F2F4F0)</td>
<td>240</td>
</tr>
</tbody>
</table>

* The data lines are hexadecimal representations of codes for characters. Each two hexadecimal digits correspond to one byte of binary data, and each byte corresponds to one character value.

---

### S370FIB\(w.d\) Informat

Reads integer binary (fixed-point) values, including negative values, in IBM mainframe format

**Category:** Numeric

#### Syntax

S370FIB\(w.d\)

#### Syntax Description

\(w\)

specifies the width of the input field.
**Default:** 4  
**Range:** 1–8

\( d \)

optionally specifies the power of 10 by which to divide the value.  
**Range:** 0–10

**Details**

The `S370FIBw.d` informat reads integer binary (fixed-point) values that are stored in IBM mainframe format, including negative values that are represented in two's complement notation. `S370FIBw.d` reads integer binary values with consistent results if the values are created in the same type of operating environment that you use to run SAS.

Use `S370FIBw.d` for integer binary data that are created in IBM mainframe format for reading in other operating environments.

*Note:* Different operating environments store integer binary values in different ways. This concept is called byte ordering. For a detailed discussion about byte ordering, see “Byte Ordering for Integer Binary Data on Big Endian and Little Endian Platforms” on page 935.

**Comparisons**

- If you use SAS on an IBM mainframe, `S370FIBw.d` and `IBw.d` are identical.
- `S370FPIBw.d`, `S370FIBUw.d`, and `S370FIBw.d` are used to read big endian integers in any operating environment.

To view a table that shows the type of informat to use with big endian and little endian integers, see Table 5.1 on page 936.

To view a table that compares integer binary notation in several programming languages, see Table 5.2 on page 936.

**Examples**

You can use the INPUT statement and specify the `S370FIB` informat. However, this example uses the informat with the INPUT function, where the binary input value is described by using a hex literal.

```sas
x=input('0080',s370fib2.);
```

<table>
<thead>
<tr>
<th>If the SAS Statement is</th>
<th>The Result is</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>put x=;</code></td>
<td>128</td>
</tr>
</tbody>
</table>
See Also

Informats:
“S370FIBUw.d Informat” on page 1022
“S370FPBw.d Informat” on page 1025

S370FIBUw.d Informat

Reads unsigned integer binary (fixed-point) values in IBM mainframe format

Category: Numeric

Syntax

S370FIBUw.d

Syntax Description

\( w \)

specifies the width of the input field.

Default: 4
Range: 1–8

\( d \)

optionally specifies the power of 10 by which to divide the value. SAS uses the \( d \) value even if the data contain decimal points.

Range: 0–10

Details

The S370FIBUw.d informat reads unsigned integer binary (fixed-point) values that are stored in IBM mainframe format, including negative values that are represented in two’s complement notation. Unsigned integer binary values are the same as integer binary values, except that all values are treated as positive. S370FIBUw.d reads integer binary values with consistent results if the values are created in the same type of operating environment that you use to run SAS.

Use S370FIBUw.d for unsigned integer binary data that are created in IBM mainframe format for reading in other operating environments.

Note: Different operating environments store integer binary values in different ways. This concept is called byte ordering. For a detailed discussion about byte ordering, see “Byte Ordering for Integer Binary Data on Big Endian and Little Endian Platforms” on page 935.

Comparisons

- The S370FIBUw.d informat is equivalent to the COBOL notation PIC 9(\( n \)) BINARY, where \( n \) is the number of digits.
The S370FIBUw.d and S370FPIBw.d informats are identical.

S370FPIBw.d, S370FIBUw.d, and S370FIBw.d are used to read big endian integers in any operating environment.

To view a table that shows the type of informat to use with big endian and little endian integers, see Table 5.1 on page 936.

To view a table that compares integer binary notation in several programming languages, see Table 5.2 on page 936.

**Examples**

You can use the INPUT statement and specify the S370FIBU informat. However, these examples use the informat with the INPUT function, where binary input values are described by using a hex literal.

```
x=input('7F'x,s370fibul.);
y=input('F6'x,s370fibul.);
```

<table>
<thead>
<tr>
<th>When the SAS Statement is ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>put x=;</td>
<td>127</td>
</tr>
<tr>
<td>put y=;</td>
<td>246</td>
</tr>
</tbody>
</table>

**See Also**

Informats:

“S370FIBw.d Informat” on page 1020
“S370FPIBw.d Informat” on page 1025

---

**S370FPDw.d Informat**

Reads packed data in IBM mainframe format

**Category:** Numeric

**Syntax**

S370FPDw.d
Syntax Description

\[ w \]
specifies the width of the input field.

- **Default:** 1
- **Range:** 1–16

\[ d \]
optionally specifies the power of 10 by which to divide the value.

- **Range:** 0–10

Details

Packed decimal data contain two digits per byte, but only one digit in the input field represents the sign. The last half of the last byte indicates the sign: a C or an F for positive numbers and a D for negative numbers.

Use S370FPDU to read packed decimal data from IBM mainframe files on other operating environments.

Comparisons

- If you use SAS on an IBM mainframe, the S370FPDU and the PDUD informats are identical.
- The following table compares the equivalent packed decimal notation by programming language:

<table>
<thead>
<tr>
<th>Language</th>
<th>Packed Decimal Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAS</td>
<td>S370FIB4.</td>
</tr>
<tr>
<td>PL/I</td>
<td>FIXED DEC(7,0)</td>
</tr>
<tr>
<td>COBOL</td>
<td>COMP-3 PIC 9(7)</td>
</tr>
<tr>
<td>assembler</td>
<td>PL4</td>
</tr>
</tbody>
</table>

S370FPDUw.d Informat

Reads unsigned packed decimal data in IBM mainframe format

- **Category:** Numeric

Syntax

S370FPDUw.d
Syntax Description

\( w \)

specifies the width of the input field.

Default: 1

Range: 1–16

\( d \)

optionally specifies the power of 10 by which to divide the value.

Range: 0–10

Details

Packed decimal data contain two digits per byte. The last half of the last byte, which indicates the sign for signed packed data, is always F for unsigned packed data.

Use S370FPDU\( w.d \) on other operating environments to read unsigned packed decimal data from IBM mainframe files.

Comparisons

- The S370FPDU\( w.d \) informat is similar to the S370FPD\( w.d \) informat except that the S370FPDU\( w.d \) informat rejects all sign digits except F.
- The S370FPDU\( w.d \) informat is equivalent to the COBOL notation PIC 9(\( n \)) PACKED-DECIMAL, where the \( n \) value is the number of digits.

Examples

```
input @1 x $370fpdu3.;
```

When the Data Line* = ... The Result is ...

<table>
<thead>
<tr>
<th>Data Line</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345F</td>
<td>12345</td>
</tr>
</tbody>
</table>

* The data line is a hexadecimal representation of a binary number that is stored in packed decimal form. Each two hexadecimal digits correspond to one byte of binary data, and each byte corresponds to one column of the input field.

S370FPIB\( w.d \) Informat

Reads positive integer binary (fixed-point) values in IBM mainframe format

Category: Numeric

Syntax

S370FPIB\( w.d \)
**Syntax Description**

\[ w \]

specifies the width of the input field.

**Default:** 4  
**Range:** 1–8

\[ d \]

optionally specifies the power of 10 by which to divide the value.  
**Range:** 0–10

**Details**

Positive integer binary values are the same as integer binary values, except that all values are treated as positive. S370FPIB\(w,d\) reads integer binary values with consistent results if the values are created in the same type of operating environment that you use to run SAS.

Use S370FPIB\(w,d\) for positive integer binary data that are created in IBM mainframe format for reading in other operating environments.

*Note:* Different operating environments store integer binary values in different ways. This concept is called byte ordering. For a detailed discussion about byte ordering, see “Byte Ordering for Integer Binary Data on Big Endian and Little Endian Platforms” on page 935.

**Comparisons**

- If you use SAS on an IBM mainframe, S370FPIB\(w,d\) and PIB\(w,d\) are identical.
- S370FPIB\(w,d\), S370FIBU\(w,d\), and S370FIB\(w,d\) are used to read big endian integers in any operating environment.

  To view a table that shows the type of informat to use with big endian and little endian integers, see Table 5.1 on page 936.

  To view a table that compares integer binary notation in several programming languages, see Table 5.2 on page 936.

**Examples**

You can use the INPUT statement and specify the S370FPIB informat. However, this example uses the informat with the INPUT function, where the binary input value is described using a hex literal.

\[ x=\text{input}('0100'x,s370fpib2.); \]

<table>
<thead>
<tr>
<th>When the SAS Statement is ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>put x=</code>;</td>
<td>256</td>
</tr>
</tbody>
</table>
See Also

Informats:
“S370FIBw.d Informat” on page 1020
“S370FIBUw.d Informat” on page 1022

S370FRBw.d Informat

Reads real binary (floating-point) data in IBM mainframe format

Category: Numeric

Syntax

S370FRBw.d

Syntax Description

\(w\)

specifies the width of the input field.

Default: 6

Range: 2–8

\(d\)

optionally specifies the power of 10 by which to divide the value.

Range: 0–10

Details

Real binary values are represented in two parts: a mantissa that gives the value, and an exponent that gives the value's magnitude.

Use S370FRBw.d to read real binary data from IBM mainframe files on other operating environments.

Comparisons

- If you use SAS on an IBM mainframe, S370FRBw.d and RBw.d are identical.
- The following table shows the equivalent real binary notation for several programming languages:

<table>
<thead>
<tr>
<th>Language</th>
<th>4 Bytes</th>
<th>8 Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL/I</td>
<td>FLOAT BIN(21)</td>
<td>FLOAT BIN(53)</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>REAL*4</td>
<td>REAL*8</td>
</tr>
</tbody>
</table>
### Real Binary Notation

<table>
<thead>
<tr>
<th>Language</th>
<th>4 Bytes</th>
<th>8 Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>COBOL</td>
<td>COMP-1</td>
<td>COMP-2</td>
</tr>
<tr>
<td>assembler</td>
<td>E</td>
<td>D</td>
</tr>
<tr>
<td>C</td>
<td>float</td>
<td>double</td>
</tr>
</tbody>
</table>

**See Also**

Informat:

“RB\(w.d\) Informat” on page 1013

---

### S370FZ\(w.d\) Informat

Reads zoned decimal data in IBM mainframe format

**Category:** Numeric

**Syntax**

S370FZ\(w.d\)

**Syntax Description**

\(w\)

specifies the width of the input field.

- **Default:** 8
- **Range:** 1–32

\(d\)

optionally specifies the power of 10 by which to divide the value. If the data contain decimal points, the \(d\) value is ignored.

- **Range:** 0–10

**Details**

Zoned decimal data are similar to standard decimal data in that every digit requires one byte. However, the value’s sign is stored in the last byte, along with the last digit. Use S370FZ\(w.d\) on other operating environments to read zoned decimal data from IBM mainframe files.

**Comparisons**

- If you use SAS on an IBM mainframe, S370FZ\(w.d\) and Z\(dw.d\) are identical.
The following table shows the equivalent zoned decimal notation for several programming languages:

<table>
<thead>
<tr>
<th>Language</th>
<th>Zoned Decimal Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAS</td>
<td>S370FZD3.</td>
</tr>
<tr>
<td>PL/I</td>
<td>PICTURE'99T'</td>
</tr>
<tr>
<td>COBOL</td>
<td>PIC S9(3) DISPLAY</td>
</tr>
<tr>
<td>assembler</td>
<td>ZL3</td>
</tr>
</tbody>
</table>

**Examples**

```
input @1 x s370fzd3.;
```

<table>
<thead>
<tr>
<th>When the Data Line* = ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1F2C3</td>
<td>123</td>
</tr>
<tr>
<td>F1F2D3</td>
<td>-123</td>
</tr>
</tbody>
</table>

* The data line contains a hexadecimal representation of a binary number stored in zoned decimal format on an IBM mainframe operating environment. Each two hexadecimal digits correspond to one byte of binary data, and each byte corresponds to one column of the input field.

**See Also**

Informat:

“ZDw.d Informat” on page 1048

**S370FZDLw.d Informat**

Reads zoned decimal leading-sign data in IBM mainframe format

**Category:** Numeric

**Syntax**

`S370FZDLw.d`
Syntax Description

\( w \)

specifies the width of the input field.

**Default:** 8  
**Range:** 1–32

\( d \)

optionally specifies the power of 10 by which to divide the value.

**Range:** 0–10

Details

Use S370FZDL\( w.d \) on other operating environments to read zoned decimal data from IBM mainframe files.

Comparisons

- Zoned decimal leading-sign data is similar to standard zoned decimal data except that the sign of the value is stored in the first byte of zoned decimal leading-sign data, along with the first digit.
- The S370FZDL\( w.d \) informat is equivalent to the COBOL notation PIC S9(\( n \)) DISPLAY SIGN LEADING, where the \( n \) value is the number of digits.

Examples

```
input @1 x s370fzd13.;
```

<table>
<thead>
<tr>
<th>Data Line</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>C1F2F3</code></td>
<td>123</td>
</tr>
<tr>
<td><code>D1F2F3</code></td>
<td>-123</td>
</tr>
</tbody>
</table>

* The data lines contain a hexadecimal representation of a binary number stored in zoned decimal format on an IBM mainframe operating environment. Each two hexadecimal digits correspond to one byte of binary data, and each byte corresponds to one column of the input field.

S370FZDS\( w.d \) Informat

Reads zoned decimal separate leading-sign data in IBM mainframe format

**Category:** Numeric
Syntax
S370FZDS\textit{w.d}

Syntax Description

\textit{w}

specifies the width of the input field.

Default: 8

Range: 2–32

\textit{d}

optionally specifies the power of 10 by which to divide the value.

Range: 0–10

Details

Use S370FZDS\textit{w.d} on other operating environments to read zoned decimal data from IBM mainframe files.

Comparisons

- Zoned decimal separate leading-sign data is similar to standard zoned decimal data except that the sign of the value is stored in the first byte of zoned decimal leading sign data, and the first digit of the value is stored in the second byte.
- The S370FZDS\textit{w.d} informat is equivalent to the COBOL notation PIC S9(\textit{n}) DISPLAY SIGN LEADING SEPARATE, where the \textit{n} value is the number of digits.

Examples

input @1 x s370fzds4.;

<table>
<thead>
<tr>
<th>When the Data Line* = ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>4EF1F2F3</td>
<td>123</td>
</tr>
<tr>
<td>60F1F2F3</td>
<td>-123</td>
</tr>
</tbody>
</table>

* The data line contains a hexadecimal representation of a binary number that is stored in zoned decimal format on an IBM mainframe operating environment. Each two hexadecimal digits correspond to one byte of binary data, and each byte corresponds to one column of the input field.

S370FZDT\textit{w.d} Informat

Reads zoned decimal separate trailing-sign data in IBM mainframe format

Category: Numeric
Syntax
\texttt{S370FZDT}w.d

Syntax Description

\textit{w}
\begin{itemize}
  \item specifies the width of the input field.
  \item Default: 8
  \item Range: 2–32
\end{itemize}

\textit{d}
\begin{itemize}
  \item optionally specifies the power of 10 by which to divide the value.
  \item Range: 0–10
\end{itemize}

Details
Use \texttt{S370FZDT}w.d on other operating environments to read zoned decimal data from IBM mainframe files.

Comparisons
- Zoned decimal separate trailing-sign data are similar to zoned decimal separate leading-sign data except that the sign of the value is stored in the last byte of zoned decimal separate trailing-sign data.
- The \texttt{S370FZDT}w.d informat is equivalent to the COBOL notation PIC S9(n) DISPLAY SIGN TRAILING SEPARATE, where the \textit{n} value is the number of digits.

Examples

\begin{verbatim}
input @1 x s370fzdt4.;
\end{verbatim}

<table>
<thead>
<tr>
<th>When the Data Line* = ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{F1F2F34E}</td>
<td>123</td>
</tr>
<tr>
<td>\texttt{F1F2F360}</td>
<td>-123</td>
</tr>
</tbody>
</table>

* The data line contains a hexadecimal representation of a binary number that is stored in zoned decimal format on an IBM mainframe operating environment. Each two hexadecimal digits correspond to one byte of binary data, and each byte corresponds to one column of the input field.

\textbf{S370FZDU}w.d Informat

Reads unsigned zoned decimal data in IBM mainframe format
Category: Numeric

Syntax
S370FZDUw.d

Syntax Description

\( w \)
specifies the width of the input field.

Default: 8  
Range: 1–32

\( d \)
optionally specifies the power of 10 by which to divide the value.

Range: 0–10

Details

Use S370FZDUw.d on other operating environments to read unsigned zoned decimal data from IBM mainframe files.

Comparisons

- The S370FZDUw.d informat is similar to the S370FZDw.d informat except that the S370FZDUw.d informat rejects all sign digits except F.
- The S370FZDUw.d informat is equivalent to the COBOL notation PIC 9(n) DISPLAY, where the \( n \) value is the number of digits.

Examples

```
input @1 x s370fzdu3.;
```

<table>
<thead>
<tr>
<th>When the Data Line* = ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>----+----</td>
<td>123</td>
</tr>
<tr>
<td>F1F2F3</td>
<td>123</td>
</tr>
</tbody>
</table>

* The data line contains a hexadecimal representation of a binary number that is stored in zoned decimal format on an IBM mainframe operating environment. Each two hexadecimal digits correspond to one byte of binary data, and each byte corresponds to one column of the input field.

---

SHRSTAMPw. Informat

Reads date and time values of SHR records
Category: Date and Time

Syntax

\texttt{SHRSTAMPw.}

Syntax Description

\texttt{w}

specifies the width of the input field.

\textbf{Requirement:} \texttt{w} must be 8 because packed decimal date and time values in SHR records contain eight bytes of information: four bytes of date data that are followed by four bytes of time data.

Details

The \texttt{SHRSTAMPw.} informat reads packed decimal date and time values of SHR records that are produced by IBM mainframe environments and converts the date and time values to SAS datetime values.

The general form of the date and time information in an SHR record in hexadecimal notation is \texttt{ccydddFhhmmsssth}, where

\textit{ccy}

is the two byte representation of the year. The \textit{cc} portion is the one byte representation of a two-digit integer that represents the century. The \textit{yy} portion is the one byte representation of two digits that correspond to the year.

The \textit{cc} portion is the century indicator where 00 indicates 19yy, 01 indicates 20yy, 02 indicates 21yy and so on. A hex year value of 0115 is equal to the year 2015.

\textit{ddd}

is the one-and-a-half bytes that contain three digits that correspond to the day of the year.

\textit{F}

is the half byte that contains all binary 1s.

\textit{hh}

is the one byte representation of two digits that correspond to the hour of the day.

\textit{mm}

is the one byte representation of two digits that correspond to minutes.

\textit{ss}

is the one byte representation of two digits that correspond to seconds.

\textit{th}

is the one byte representation of two digits that correspond to a hundredth of a second.

The \texttt{SHRSTAMPw.} informat enables you to read, on any operation environment, packed decimal date and time values from files that are created on an IBM mainframe.
Examples

```
input begin shrstamp8.;
```

<table>
<thead>
<tr>
<th>When the Data Line* = ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0097239F12403576</td>
<td>1188304835.8</td>
</tr>
</tbody>
</table>

* The data line is a hexadecimal representation of a packed decimal date and time value that is stored as it would appear in an SHR record. Each byte occupies one column of the input field. The result is a SAS datetime value that corresponds to Aug. 27, 1997 12:40:36 PM.

**SMFSTAMPw. Informat**

Reads time and date values of SMF records

**Category:** Date and Time

**Syntax**

```
SMFSTAMPw.
```

**Syntax Description**

```
\( w \)
```

specifies the width of the input field.

**Requirement:** \( w \) must be 8 because time and date values in SMF records contain eight bytes of information: four bytes of time data that are followed by four bytes of date data.

**Tip:** The time portion of an SMF record is a four-byte integer binary number that represents time as the number of hundredths of a second past midnight.

**Details**

The SMFSTAMPw. informat reads integer binary time values and packed decimal date values of SMF records that are produced by IBM mainframe systems and converts the time and date values to SAS datetime values.

The date portion of an SMF record in hexadecimal notation is \( ccyydddF \), where

- \( cc \) is the one-byte representation of two digits that correspond to the century.
- \( yy \) is the one-byte representation of two digits that correspond to the year.
is the one-and-a-half bytes that contain three digits that correspond to the day of the year.

F

is the half byte that contains all binary 1s.

The SMFSTAMPw. informat enables you to read, on any operating environment, integer binary time values and packed decimal date values from files that are created on an IBM mainframe.

Examples

```
input begin smfstamp8.;
```

When the Data Line* = ... The Result is ...

<table>
<thead>
<tr>
<th>Data Line</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0058DC0C0098200F</td>
<td>1216483835</td>
</tr>
</tbody>
</table>

* The data line is a hexadecimal representation of a binary time and date value that is stored as it would appear in an SMF record. Each byte occupies one column of the input field. The result is a SAS datetime value that corresponds to July 19, 1998 4:10:35 PM.

STIMERw. Informat

Reads time values and determines whether the values are hours, minutes, or seconds; reads the output of the STIMER system option

Category: Date and Time

Syntax

```
STIMERw.
```

Syntax Description

w

specifies the width of the input field.

Details

The STIMER informat reads performance statistics that the STIMER system option writes to the SAS log.
The informat reads time values and determines whether the values are hours, minutes, or seconds based on the presence of decimal points and colons:

- If no colon is present, the value is the number of seconds.
- If a single colon is present, the value before the colon is the number of minutes. The value after the colon is the number of seconds.
- If two colons are present, the sequence of time is hours, minutes, and then seconds.

In all cases, the result is a SAS time value.

The input values for STIMER must be in one of the following forms:

- `ss`
- `ss.ss`
- `mm:ss`
- `mm:ss.ss`
- `hh:mm:ss`
- `hh:mm:ss.ss`

where

- `ss` is an integer that represents the number of seconds.
- `mm` is an integer that represents the number of minutes.
- `hh` is an integer that represents the number of hours.

---

**TIMEw. Informat**

**Reads hours, minutes, and seconds in the form `hh:mm:ss.ss`**

**Category:** Date and Time

**Syntax**

`TIMEw.`

**Syntax Description**

`w`

- specifies the width of the input field.

  **Default:** 8

  **Range:** 5–32

**Details**

Time values must be in the form `hh:mm:ss.ss`, where
\[ hh \]
\[ is \ an\ integer. \]

\[ mm \]
\[ is \ the\ number\ of\ minutes\ that\ range\ from\ 00\ through\ 59. \]

\[ ss.ss \]
\[ is \ the\ number\ of\ seconds\ ranging\ from\ 00\ through\ 59\ with\ the\ fraction\ of\ a\ second\ following\ the\ decimal\ point. \]

Separate \( hh \), \( mm \), and \( ss.ss \) with a special character. If you do not enter a value for seconds, SAS assumes a value of 0.

The stored value is the total number of seconds in the time value.

**Examples**

```plaintext
input begin time10.;
```

<table>
<thead>
<tr>
<th>When the Data Line = ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>11:23:07.4</td>
<td>40987.4</td>
</tr>
</tbody>
</table>

The TIME informat can read time values with AM or PM in the value.

```plaintext
input begin time8.;
```

<table>
<thead>
<tr>
<th>Data Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:13 PM</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>47580.0</td>
</tr>
</tbody>
</table>

**See Also**

Formats:

“HHMMw.d Format” on page 153
“HOURw.d Format” on page 155
“MMSSw.d Format” on page 166
“TIMEw.d Format” on page 209

Functions:

“HOUR Function” on page 579
“MINUTE Function” on page 646
“SECOND Function” on page 820
“TIME Function” on page 856
**TODSTAMP<sub>w</sub>. Informat**

Reads an eight-byte time-of-day stamp  
**Category:** Date and Time

**Syntax**  
TODSTAMP<sub>w</sub>.  

**Syntax Description**  

<sub>w</sub>  
specifies the width of the input field.  
**Requirement:** <sub>w</sub> must be 8 because the OS TIME macro or the STCK instruction on IBM mainframes each return an eight-byte value.

**Details**  
The TODSTAMP<sub>w</sub>. informat reads time-of-day clock values that are produced by IBM mainframe operating systems and converts the clock values to SAS datetime values.  
If the time-of-day value is all 0s, TODSTAMP<sub>w</sub>. results in a missing value.  
Use TODSTAMP<sub>w</sub>. on other operating environments to read time-of-day values that are produced by an IBM mainframe.

**Examples**

```
input btime todstamp8.;
```

<table>
<thead>
<tr>
<th>When the Data Line* = ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>B361183D5FB80000</td>
<td>1262303998</td>
</tr>
</tbody>
</table>

* The data line is a hexadecimal representation of a binary, 8-byte time-of-day clock value. Each byte occupies one column of the input field. The result is a SAS datetime value that corresponds to December 31, 1999, 11:59:58 PM.

**TRAILSGN<sub>w</sub>. Informat**

Reads a trailing plus (+) or minus (-) sign  
**Category:** Numeric
Syntax
TRAILSGNW.

Syntax Description

\( w \)
specifies the width of the input field.

**Default:** 6

**Range:** 1–32

Details
If the data contains a decimal point, then the TRAILSGN informat honors the number of decimal places that are in the input data.

Examples

```plaintext
input x trailsgn8.;
```

<table>
<thead>
<tr>
<th>When the data line is...</th>
<th>The result is...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1+</td>
<td>1</td>
</tr>
<tr>
<td>1-</td>
<td>-1</td>
</tr>
<tr>
<td>1.2</td>
<td>1.2</td>
</tr>
<tr>
<td>1.2+</td>
<td>1.2</td>
</tr>
<tr>
<td>1.2-</td>
<td>-1.2</td>
</tr>
</tbody>
</table>
```

TUw. Informat

Reads timer units

**Category:** Date and Time

Syntax
TUw.
Syntax Description

\( w \)

specifies the width of the input field.

**Requirement:** \( w \) must be 4 because the OS TIME macro returns a four-byte value.

Details

The TU\( w \). informat reads timer unit values that are produced by IBM mainframe operating environments and converts the timer unit values to SAS time values.

There are exactly 38,400 software timer units per second. The low-order bit in a timer unit value represents approximately 26.041667 microseconds.

Use the TU\( w \). informat to read timer unit values that are produced by an IBM mainframe on other operating environments.

Examples

```sas
input btime tu4.;
```

When the Data Line* = ...  The Result is ...

<table>
<thead>
<tr>
<th>Data Line</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>8FC7A9BC</td>
<td>62818.41563</td>
</tr>
</tbody>
</table>

* The data line is a hexadecimal representation of a binary, four-byte timer unit value. Each byte occupies one column of the input field. The result is a SAS time value that corresponds to 5:26:58.41 PM.

**VAXRB\( w \).d Informat**

Reads real binary (floating-point) data in VMS format

Category: Numeric

Syntax

VAXRB\( w \).d

Syntax Description

\( w \)

specifies the width of the input field.

**Default:** 4

**Range:** 2–8
$d$

optionally specifies the power of 10 by which to divide the value.

**Range:** 0–10

**Details**

Use the VAXRB$w.d$ informat to read floating-point data from VMS files on other operating environments.

**Comparisons**

If you use SAS that is running under VMS, the VAXRB$w.d$ and the RB$w.d$ informats are identical.

**See Also**

Informat:

“RB$w.d$ Informat” on page 1013

---

**$w.d$ Informat**

Reads standard numeric data

**Category:** Numeric

**Alias:** BEST$w.d$, Dw.$d$, Ew.$d$, Fw.$d$

**Syntax**

$w.d$

**Syntax Description**

$w$

specifies the width of the input field.

**Range:** 1–32

$d$

optionally specifies the power of 10 by which to divide the value. If the data contain decimal points, the $d$ value is ignored.

**Range:** 0–31

**Details**

The $w.d$ informat reads numeric values that are located anywhere in the field. Blanks can precede or follow a numeric value with no effect. A minus sign with no separating blank should immediately precede a negative value. The $w.d$ informat reads values with decimal points and values in scientific E-notation, and it interprets a single period as a missing value.
Comparisons

- The \( w.d \) informat is identical to the BZ\( w.d \) informat, except that the \( w.d \) informat ignores trailing blanks in the numeric values. To read trailing blanks as 0s, use the BZ\( w.d \) informat.

- The \( w.d \) informat can read values in scientific E-notation exactly as the E\( w.d \) informat does.

Examples

```
input @1 x 6. @10 y 6.2;
put x @7 y;
```

<table>
<thead>
<tr>
<th>When the Data Line = ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>23 2300</td>
<td>23 23</td>
</tr>
<tr>
<td>23 -2300</td>
<td>23 -23</td>
</tr>
<tr>
<td>23.0 23.</td>
<td>23 23</td>
</tr>
<tr>
<td>2.3E1 2.3</td>
<td>23 2.3</td>
</tr>
<tr>
<td>-23 0</td>
<td>-23 0</td>
</tr>
</tbody>
</table>

WEEKU\( w \). Informat

Reads the format of the number-of-week value within the year and returns a SAS date value by using the U algorithm

Category: Date and Time


WEEKV\( w \). Informat

Reads the format of the number-of-week value within the year and returns a SAS date value using the V algorithm

Category: Date and Time

**WEEKWw. Informat**

Reads the format of the number-of-week value within the year and returns a SAS date value using the W algorithm

*Category:* Date and Time

*See:* The WEEKW informat in *SAS National Language Support (NLS): User’s Guide*

---

**YENw.d Informat**

Removes embedded yen signs, commas, and decimal points

*Category:* Numeric

*See:* The YEN informat in *SAS National Language Support (NLS): User’s Guide*

---

**YYMMDDw. Informat**

Reads date values in the form *yymmdd* or *yyyyymmdd*

*Category:* Date and Time

**Syntax**

**YYMMDDw.**

**Syntax Description**

**w**

specifies the width of the input field.

*Default:* 6

*Range:* 6–32

**Details**

The date values must be in the form *yymmdd* or *yyyyymmdd*, where

- *yy* or *yyy*
  - is a two-digit or four-digit integer that represents the year.

- *mm*
  - is an integer from 01 through 12 that represents the month of the year.

- *dd*
is an integer from 01 through 31 that represents the day of the month.

You can separate the year, month, and day values by blanks or by special characters. However, if delimiters are used, place them between all the values. You can also place blanks before and after the date. Make sure the width of the input field allows space for blanks and special characters.

Note: SAS interprets a two-digit year as belonging to the 100-year span that is defined by the YEARCUTOFF= system option.

Examples

input calendar_date yymmdd10.;

<table>
<thead>
<tr>
<th>When the Data Line = …</th>
<th>The Result is …</th>
</tr>
</thead>
<tbody>
<tr>
<td>050316</td>
<td>16511</td>
</tr>
<tr>
<td>05/03/16</td>
<td>16511</td>
</tr>
<tr>
<td>05 03 16</td>
<td>16511</td>
</tr>
<tr>
<td>2005–03–16</td>
<td>16511</td>
</tr>
</tbody>
</table>

See Also

Formats:
“DATEw. Format” on page 118
“DDMMYYw. Format” on page 123
“MMDDYYw. Format” on page 162
“YYMMDDw. Format” on page 229

Functions:
“DAY Function” on page 481
“MDY Function” on page 643
“MONTH Function” on page 655
“YEAR Function” on page 916

Informats:
“DATEw. Informat” on page 979
“DDMMYYw. Informat” on page 982
“MMDDYYw. Informat” on page 993

System Option:
“YEARCUTOFF= System Option” on page 1630

YYMMNw. Informat

Reads date values in the form yyyy-mm or yymm
Category: Date and Time

Syntax

YYMMN<\textsubscript{\textit{w}}>.  

Syntax Description

\textit{w}  

specifies the width of the input field.  

Default: 4  

Range: 4–6

Details

The date values must be in the form \textit{yyyy}mm or \textit{ym}m, where

\textit{yy} or \textit{yyyy}  

is a two-digit or four-digit integer that represents the year.

mm  

is a two-digit integer that represents the month.

The \textit{N} in the informat name must be used and indicates that you cannot separate the year and month values by blanks or by special characters. SAS automatically adds a day value of 01 to the value to make a valid SAS date variable.

\textit{Note}: SAS interprets a two-digit year as belonging to the 100-year span that is defined by the YEARCUTOFF= system option.

Examples

\begin{verbatim}
input date1 yyymnn6.;
\end{verbatim}

When the Data Line = … The Result is …

\begin{verbatim}
200508  16649
\end{verbatim}

See Also

Formats:

“DATE<\textit{w}. Format” on page 118  
“DDMMYY<\textit{w}. Format” on page 123  
“YYMMD<\textit{w}. Format” on page 229  
“YYMM<\textit{w}. Format” on page 226  
“YYMON<\textit{w}. Format” on page 232
YYQw. Informat

Reads quarters of the year in the form yyQq or yyyyQq

Category: Date and Time

Syntax

YYQw.

Syntax Description

w

specifies the width of the input field.

Default:  6 (For SAS version 6, the default is 4.)

Range:  4–32 (For SAS version 6, the range is 4–6.)

Details

The quarter must be in the form yyQq or yyyyQq, where

yy or yyyy

is an integer that represents the two-digit or four-digit year.

q

is an integer (1, 2, 3, or 4) that represents the quarter of the year. You can also represent the quarter as 01, 02, 03, or 04.

The letter Q must separate the year value and the quarter value. The year value, the letter Q, and the quarter value cannot be separated by blanks. A value that is read with YYQw. produces a SAS date value that corresponds to the first day of the specified quarter.

Note: SAS interprets a two-digit year as belonging to the 100-year span that is defined by the YEARCUTOFF= system option. △
### Examples

```plaintext
input quarter yyq9.;
```

<table>
<thead>
<tr>
<th>When the Data Line = ...</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>05Q2</td>
<td>16527</td>
</tr>
<tr>
<td>05Q02</td>
<td>16527</td>
</tr>
<tr>
<td>2005Q02</td>
<td>16527</td>
</tr>
</tbody>
</table>

### See Also

Functions:
- “QTR Function” on page 760
- “YEAR Function” on page 916
- “YYQ Function” on page 920

System Option:
- “YEARCUTOFF= System Option” on page 1630

---

### ZD\textit{w}.\textit{d} Informat

**Reads zoned decimal data**

**Category:** Numeric

**See:** \textit{ZDw.d} Informat in the documentation for your operating environment.

### Syntax

\textit{ZDw.d}

### Syntax Description

- \textit{w}
  - specifies the width of the input field.
  - **Default:** 1
  - **Range:** 1–32

- \textit{d}
  - optionally specifies the power of 10 by which to divide the value.
  - **Range:** 1–31
Details

The ZDw.d informat reads zoned decimal data in which every digit requires one byte and in which the last byte contains the value’s sign along with the last digit.

Note: Different operating environments store zoned decimal values in different ways. However, ZDw.d reads zoned decimal values with consistent results if the values are created in the same type of operating environment that you use to run SAS.

You can enter positive values in zoned decimal format from a terminal. Some keying devices enable you to enter negative values by overstriking the last digit with a minus sign.

Comparisons

- Like the w.d informat, the ZDw.d informat reads data in which every digit requires one byte. Use ZDVw.d or ZDw.d to read zoned decimal data in which the last byte contains the last digit and the sign.
- The ZDw.d informat functions like the ZDVw.d informat with one exception: ZDVw.d validates the input string and disallows invalid data.
- The following table compares the zoned decimal informat with notation in several programming languages:

<table>
<thead>
<tr>
<th>Language</th>
<th>Zoned Decimal Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAS</td>
<td>ZD3</td>
</tr>
<tr>
<td>PL/I</td>
<td>PICTURE‘99T’</td>
</tr>
<tr>
<td>COBOL</td>
<td>DISPLAY PIC S 999</td>
</tr>
<tr>
<td>IBM 370 assembler</td>
<td>ZL3</td>
</tr>
</tbody>
</table>

Examples

```sas
input @1 x zd4.;
```

When the Data Line* = ... The Result is ...

| -----+----1  | 128     |
|-------+---------|
| F0F1F2C8 |         |

* The data line contains a hexadecimal representation of a binary number that is stored in zoned decimal format on an IBM mainframe computer system. Each byte occupies one column of the input field.

See Also

Informats:
“w.d Informat” on page 1042
“ZDVw.d Informat” on page 1050
**ZDBw.d Informat**

Reads zoned decimal data in which zeros have been left blank

Category: Numeric

See: ZDBw.d Informat in the documentation for your operating environment.

---

**Syntax**

ZDBw.d

**Syntax Description**

\[ w \]

specifies the width of the input field.

Default: 1

Range: 1–32

\[ d \]

optionally specifies the power of 10 by which to divide the value.

Range: 0–31

**Details**

The ZDBw.d informat reads zoned decimal data that are produced in IBM 1410, 1401, and 1620 form, where 0s are left blank rather than being punched.

**Examples**

```plaintext
input @1 x zdb3.;
```

When the Data Line* = ... The Result is ...

| -----+----- | 1 |
| F140C2 | 102 |

* The data line contains a hexadecimal representation of a binary number that is stored in zoned decimal form, including the codes for spaces, on an IBM mainframe operating environment. Each byte occupies one column of the input field.

---

**ZDVw.d Informat**

Reads and validates zoned decimal data
Category: Numeric

Syntax
ZDVw.d

Syntax Description

w
  specifies the width of the input field.
  Default:  1
  Range:  1–32

d
  optionally specifies the power of 10 by which to divide the value.
  Range:  1–31

Details
The ZDVw.d informat reads data in which every digit requires one byte and in which
the last byte contains the value’s sign along with the last digit. It also validates the
input string and disallows invalid data.

  ZDVw.d is dependent on the operating environment. For example, on IBM
mainframes, ZDVw.d requires an F for all high-order nibbles except the last. (In
contrast, the ZDw.d informat ignores the high-order nibbles for all bytes except those
that are associated with the sign.) The last high-order nibble accepts values ranging
from A-F, where A, C, E, and F are positive values and B and D are negative values.
The low-order nibble on IBM mainframes must be a numeric digit that ranges from 0-9,
as with ZD.

  Note: Different operating environments store zoned decimal values in different
ways. However, the ZDVw.d informat reads zoned decimal values with consistent
results if the values are created in the same type of operating environment that you use
to run SAS.

Comparisons
The ZDVw.d informat functions like the ZDw.d informat with one exception: ZDVw.d
validates the input string and disallows invalid data.

Examples

  input @1 test zdv4.;

<table>
<thead>
<tr>
<th>When the Data Line* =</th>
<th>The Result is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>F0F1F2C8</td>
<td>128</td>
</tr>
</tbody>
</table>

* The data line contains a hexadecimal representation of a binary number stored in zoned decimal
  form. The example was run on an IBM mainframe. The results may vary depending on your
  operating environment.
See Also

Informats:

“w.d Informat” on page 1042
“ZDw.d Informat” on page 1048
Definition of ARM Macros 1053
Using ARM Macros 1054
  Overview of ARM Macros 1054
  Using Variables with ARM Macros 1055
  ARM API Objects 1056
  ID Management Using ARM Macros 1056
    Complex ARM Macro Call Schemes 1059
Defining User Metrics in ARM Macros 1061
Defining Correlators in ARM Macros 1062
Enabling ARM Macro Execution 1063
  Setting the _ARMEXEC Macro Variable 1063
  Enabling ARM Macro Execution with SCL 1064
  Conditional ARM Macro Execution 1064
Setting the Macro Environment 1065
Using ARM Post-Processing Macros 1066
Troubleshooting Error Messages 1067
ARM Macros by Category 1068
Dictionary 1069
  %ARMCONV Macro 1069
  %ARMEND Macro 1070
  %ARMGTID Macro 1072
  %ARMINIT Macro 1074
  %ARMJOIN Macro 1076
  %ARMPROC Macro 1078
  %ARMSTOP Macro 1079
  %ARMSTRT Macro 1081
  %ARMUPDT Macro 1084

Definition of ARM Macros

The ARM macros provide a way to measure the performance of applications as they are executing. The macros write transaction records to the ARM log. The ARM log is an external output text file that contains the logged ARM transaction records. You insert the ARM macros into your SAS program at strategic points in order to generate calls to the ARM API function calls. The ARM API function calls typically log the time of the call and other related data to the log file. Measuring the time between these function calls yields an approximate response-time measurement.

An ARM macro is self-contained and does not affect any code surrounding it, provided that the variable name passed as an option to the ARM macro is unique. The
ARM macros are used in open code (code that is not in PROC or DATA steps) and in DATA step or SCL environments.

There are two categories of ARM macros:

- ARM macros to instrument applications
- ARM post-processing macros to use with the ARM log.

Note: The ARM macros are not part of SAS Macro Facility. They are part of the SAS ARM interface. See “Monitoring Performance Using Application Response Measurement (ARM)” in SAS Language Reference: Concepts for information about the SAS ARM interface that consists of the implementation of the ARM API as an ARM agent, ARM macros, and ARM system options.

Using ARM Macros

Overview of ARM Macros

The ARM macros invoke the ARM API function calls. The ARM macros automatically manage the returned IDs from the ARM API function calls.

The ARM API function calls are implemented in SAS software with SAS ARM macros and these function calls provide a way to measure the performance of SAS applications as they are running. For each ARM API function call, there is a corresponding macro. The following table shows the relationship among the SAS ARM macros and ARM API function calls:

<table>
<thead>
<tr>
<th>SAS ARM Macro</th>
<th>ARM API Function Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ARMINIT</td>
<td>ARM_INIT</td>
</tr>
<tr>
<td>%ARMGTID</td>
<td>ARM_GETID</td>
</tr>
<tr>
<td>%ARMSTRRT</td>
<td>ARM_START</td>
</tr>
<tr>
<td>%ARMUPDT</td>
<td>ARM_UPDATE</td>
</tr>
<tr>
<td>%ARMSTOP</td>
<td>ARM_STOP</td>
</tr>
<tr>
<td>%ARMEND</td>
<td>ARM_END</td>
</tr>
</tbody>
</table>

The SAS ARM macro invokes the ARM API function call.

The following ARM macros are available:

- “%ARMINIT Macro” on page 1074 generates a call to the ARM_INIT function call, which names the application and optionally the users of the application and initializes the ARM environment for the application. Typically, you would insert this macro in your code once.

- “%ARMGTID Macro” on page 1072 generates a call to the ARM_GETID function call, which names a transaction. Use %ARMGTID for each unique transaction in order to describe the type of transactions to be logged. A %ARMGTID is typically coded for each transaction class in an application.

- “%ARMSTRRT Macro” on page 1081 generates a call to the ARM_START function call, which signals the start of a transaction instance. Insert %ARMSTRRT before each
transaction that you want to log. Whereas %ARMGTID defines a transaction class, %ARMSTRT indicates that a transaction is executing.

"%ARMUPDT Macro" on page 1084 is an optional macro that generates a call to the ARM.UPDATE function call, which provides additional information about the progress of a transaction. Insert %ARMUPDT between %ARMSTRT and %ARMSTOP in order to supply information about the transaction that is in progress.

"%ARMSTOP Macro" on page 1079 generates a call to the ARM_STOP function call, which signals the end of a transaction instance. Insert %ARMSTOP where the transaction is known to be complete.

"%ARMEND Macro" on page 1070 generates a call to the ARM.END function call, which terminates the ARM environment and signals that the application will not make any more ARM calls.

The ARM macros permit conditional execution by setting the appropriate macro options and variables.

Some general points about the ARM macros follow:
- A general recommendation with all ARM macros is to avoid, in the program, the use of either macro variables or SAS variables beginning with the letters "_ARM."
- All macro options are keyword parameters. There are no positional parameters. Values for the macro options should be valid SAS values—that is, a SAS variable, quoted character string, numeric constant, and so on.
- ARM macros can function either inside a DATA step or in open code. Use the _ARMACRO global variable or the MACONLY=YES|NO macro option to tell the macro execution which mode is being used. For more information, see "Setting the Macro Environment" on page 1065.

Using Variables with ARM Macros

The ARM macros use variables to pass IDs and other information from one macro to another. Because the ARM macros function within the same DATA step, across DATA steps, and in open code, variables that are used by the macros can take the form of DATA step variables or macro variables as determined by the macro environment.

SAS DATA step variables are used to pass ID information between two or more ARM macros in the same DATA step. In the DATA step environment, but not in the SCL environment, DROP statements are generated for these variables so that they are not inadvertently included in any output data sets.

If ID information must be passed between two or more ARM macros across separate DATA steps, then macro global variables are used.

The following SAS DATA step variables and macro variables are considered global:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Set By</th>
<th>Used as Input by</th>
</tr>
</thead>
<tbody>
<tr>
<td>_ARMAPID</td>
<td>application ID</td>
<td>%ARMINIT</td>
<td>%ARMEND</td>
</tr>
<tr>
<td>_ARMTXID</td>
<td>transaction class ID</td>
<td>%ARMGTID</td>
<td>%ARMSTRT</td>
</tr>
<tr>
<td>_ARMSHDL</td>
<td>start handle</td>
<td>%ARMSTRT</td>
<td>%ARMUPDT, %ARMSTOP</td>
</tr>
</tbody>
</table>
### Variable Description Set By Used as Input by

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Set By</th>
<th>Used as Input by</th>
</tr>
</thead>
<tbody>
<tr>
<td>_ARMRC</td>
<td>error status</td>
<td>%ARMUPDT</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td>%ARMSTOP</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>%ARMEND</td>
<td></td>
</tr>
<tr>
<td>_ARMGLVL</td>
<td>global level indicator</td>
<td>calling program</td>
<td>all</td>
</tr>
<tr>
<td>_ARMTLVL</td>
<td>target level indicator</td>
<td>calling program</td>
<td>all</td>
</tr>
<tr>
<td>_ARMEXEC</td>
<td>global enablement</td>
<td>calling program</td>
<td>all</td>
</tr>
<tr>
<td>_ARMSCL</td>
<td>SCL code enablement</td>
<td>calling program</td>
<td>all</td>
</tr>
</tbody>
</table>

### ARM API Objects

The following three classes of objects are specified in the ARM API:

- **applications** represent the systems that you are creating, such as an inventory or order entry application. Because the SAS interface to the ARM API provides totals on a per application basis, you might want to consider this when you define the scope of your application.

- **transaction classes** specify a unit of work. You should create a transaction class for each major type of work that you want to create within an application. In concept, the transaction class is a template for the started transaction.

- **transaction instances** specify the actual start time for a unit of work. Transaction instances have response time information that is associated with them.

The ARM API uses numeric identifiers or IDs to uniquely identify the ARM objects that are input and output from the ARM macro. The three different classes of IDs that correspond to the three ARM classes are:

- application IDs
- transaction class IDs
- start handles (start time) for transaction instances.

### ID Management Using ARM Macros

These examples demonstrate how the ARM macros work. The ARM macros automatically manage application IDs, transaction IDs, and start handles. The default ID management works best in simple ARM call schemes. See “Complex ARM Macro Call Schemes” on page 1059 for more information. By default, the ARM macros use IDs that were generated from the most recent macro call. The following example demonstrates how to use all of the ARM macros:

```plaintext
/*global macro variable to indicate ARM macros are outside the data step*/
%let _armacro=1;

/* start of the application */
%arminit(appname='Sales App', appuser='userxyz');

/* define the transaction classes */
%armgtid(txnname='Sales Order', txndet='Sales Order Transaction');
/* more arm_getid function calls go here for different transaction classes */
```
/* start of a transaction */
%armstrt;

data _null_;  
/* place the actual transaction code here */
/* update the status of the transaction as it is running */
%armupdt(data='Sales transaction still running...',maconly=no);
run;

/* place the actual transaction stop here */
/* the transaction has stopped */
%armstop(status=0);

/* end of the application */
%armend;

All ID management is performed by the macros without requiring the calling
program to track IDs. Each macro in the previous example uses the most recently
generated ID values from previous macros. The following example is identical, but the
comments explain the passing of IDs in more detail:

%let _armacro=1;

/*
 * This %arminit macro will generate both a SAS
 * variable and global macro variable by the name
 * of _armapid and set it in the ID that is returned
 * from the arm_init() function call that is
 * wrapped by the macro.
 */
%arminit(appname='Sales App', appuser='userxyz');

/*
 * This %armgtid macro uses the _armapid SAS variable
 * as input to the arm_getid() function call that it wraps.
 * It also generates both a SAS variable and global macro
 * variable by the name of _armtxid and sets them to the
 * ID that is returned from the arm_getid function call that
 * it wraps.
 */
%armgtid(txnname='Sales Order', txndet='Sales Order Transaction');

/*
 * Because we are still in the same DATA step, the %armstrt
 * macro below will use the _armtxid SAS variable that is
 * generated from the previous %armgtid macro as input
 * to the arm_start() function call that it wraps. It
 * also generates an _armshdl variable.
 */
%armstrt;

/*
 * The %armupdt call below uses the _armshdl SAS variable
* that is generated from the previous %armstrt macro.
 */
%armupd(data=‘Sales transaction still running...’);

/*
 * The armstop call also uses the same _armshdl SAS
 * variable from the %armstrt.
 */
%armstop(status=0);

/*
 * The %armend call uses the _armapid SAS variable
 * generated by the %arminit macro earlier to end
 * the application.
 */
%armend;

run;

You can code the ARM macros across different DATA steps as follows and achieve the
same results:

data _null_
   /* note the end of the application */
   %arminit(appname=‘Sales App’, appuser=‘userxyz’);
run;

data _null_
   %armgtid(txnname=‘Sales Order’, txndet=‘Sales Order Transaction’);
   /* more arm_getid calls go here for different transaction classes */
run;

data _null_
   /* note the start of the transaction */
   %armstrt;

   /* place the actual transaction here */
run;

data _null_
   /* update the status of the transaction as it is running */
   %armupd(data=‘Sales transaction still running...’);
run;

data _null_
   /* place the actual transaction stop here */
   /* note that the transaction has stopped */
   %armstop(status=0);
run;

data _null_
   /* note the end of the application */
   %armend;
run;
The end result is the same as in the first example, except that the macros are using the generated macro variables rather than the SAS variables for passing IDs.

**Complex ARM Macro Call Schemes**

Allowing the macros to automatically use the global variables in basic scenarios simplifies coding. However, macros that use global variables can lead to misleading results in more complicated scenarios when you attempt to monitor concurrent applications or transactions as follows:

```sas
data _null_; %arminit(appname='App 1',getid=yes,txnname='txn 1'); run;

/* start transaction instance 1*/
data _null_; %armstrt; run;

/* start transaction instance 2 */
data _null_; %armstrt; run;

/* WRONG! This assumes that the %armupd is updating
* the first transaction. However, it is actually updating the
* second transaction instance because _armshdl contains the value
* from the last macro call that was executed, which is the second
* transaction.
*/
data _null_; %armupd(data='txn instance 1 still running...'); run;

To save the IDs use the *var options (APPIDVAR=, TXNIDVAR=, and SHDLVAR=) to pass or return the IDs in your own named variables. Here is an example that uses the SHDLVAR= option to save the start handles:

```sas
data _null_; %arminit(appname='xyz',getid=YES,txname='txn 1'); run;

/* start transaction instance 1 and save the ID using shdlvar= */
data _null_; %armstrt(shdlvar=savhd1); run;

/* start transaction instance 2 and save the ID using shdlvar= */
data _null_; /*armstrt( shdlvar=savhd12 );
run;

/* Now use the shandle= parameter after retrieving the first id. */
data _null_; %armupd(data='updating txn 1', shdlvar=savhd1); run;
```
/* Use the same technique to stop the transactions */
/* in the order they were started. */
data _null_
  %armstop(shdlvar=savhdl1);
  %armstop(shdlvar=savhdl2);
  %armend();
run;

As the previous example shows, using the *var option simplifies the code. The
previous technique is recommended for use on all ARM macro calls.
The following example demonstrates how to use all of the *var options to
automatically manage IDs for concurrent applications, transaction classes, transaction
instances, and correlated transaction instances:

data _null_
  %arminit(appname='Appl 1', appuser='userid', appidvar=app1);
  %arminit(appname='Appl 2', appuser='userid', appidvar=app2);
  %arminit(appname='Appl 3', appuser='userid', appidvar=app3);
run;

data _null_
  %armgtid(txnname='Txn 1A', txndet='Txn Class 1A', appidvar=app1,txnidvar=txn1a);
  %armgtid(txnname='Txn 1B', txndet='Txn Class 1B', appidvar=app1,txnidvar=txn1b);
  %armgtid(txnname='Txn 2A', txndet='Txn Class 2A', appidvar=app2,txnidvar=txn2a);
  %armgtid(txnname='Txn 2B', txndet='Txn Class 2B', appidvar=app2,txnidvar=txn2b);
  %armgtid(txnname='Txn 3A', txndet='Txn Class 3A', appidvar=app3,txnidvar=txn3a);
  %armgtid(txnname='Txn 3B', txndet='Txn Class 3B', appidvar=app3,txnidvar=txn3b);
run;

data _null_
  %armstrt(txnidvar=txn1a,shdlvar=sh1a);
  %armstrt(txnidvar=txn1b,shdlvar=sh1b);
  %armstrt(txnidvar=txn2a,shdlvar=sh2a);
  %armstrt(txnidvar=txn2b,shdlvar=sh2b);
  %armstrt(txnidvar=txn3a,shdlvar=sh3a);
  %armstrt(txnidvar=txn3b,shdlvar=sh3b);
run;

data _null_
  %armupd(data='Updating txn instance 1a...', shdlvar=sh1a);
  %armupd(data='Updating txn instance 1b...', shdlvar=sh1b);
  %armupd(data='Updating txn instance 2a...', shdlvar=sh2a);
  %armupd(data='Updating txn instance 2b...', shdlvar=sh2b);
  %armupd(data='Updating txn instance 3a...', shdlvar=sh3a);
  %armupd(data='Updating txn instance 3b...', shdlvar=sh3b);
run;

data _null_
%armstop(status=0, shdlvar=sh1a); %armstop(status=1, shdlvar=sh1b);
%armstop(status=0, shdlvar=sh2a); %armstop(status=1, shdlvar=sh2b);
%armstop(status=0, shdlvar=sh3a); %armstop(status=1, shdlvar=sh3b);
run;

data _null_;  
%armend(appidvar=app1);
%armend(appidvar=app2);
%armend(appidvar=app3); run;

As the previous example demonstrates, you can establish your own naming conventions to uniquely identify applications, transaction classes, and transaction instances across different DATA steps, in open code, or in SCL programs.

The macros support explicit passing of the IDs using the APPID=, TXNID=, and SHANDLE= options. These options are similar to the *var options, except that they do not retrieve values across DATA steps using macro variables. The primary use of the options is to supply numeric constants as ID values to the macros, because the *var options do not accept numeric constants.

Note: The use of APPID=, TXNID=, and SHANDLE= is not recommended for new applications. These options are maintained for compatibility with earlier ARM macro releases only. Use APPIDVAR=, TXNIDVAR=, and SHDLVAR= instead of APPID=, TXNID=, and SHANDLE=, respectively. △

Because IDs are generated by the ARM agent, to pass a numeric literal requires that you start a new SAS session. If any SAS subsystems are also functioning, you will not know what the ID will be at execution time.

The use of APPIDVAR=, TXNIDVAR=, and SHDLVAR= options is recommended when coding new applications.

---

### Defining User Metrics in ARM Macros

A metric is a counter, gauge, numeric ID, or string that you define. You specify one or more metrics for each ARM transaction class. When a start handle (instance of the transaction class) is started, updated, or stopped, the application indicates a value for the metric and writes it to the ARM log by the ARM agent.

The user metric name and user metric definition must be specified together in the %ARMGTID. METRNAM1–7= names the user metric and must be a SAS character variable or quoted literal value up to eight characters in length. METRDEF1–7= defines the output of the user-defined metric. The value of METRDEF1–6= must be one of the following:

- **COUNT32** use the counter to sum up the values over an interval. A counter can also calculate average values, maximums, and minimums per transaction, and other statistical calculations.
- **COUNT64**
- **COUNTDIV**
- **GAUGE32** use the gauge when a sum of values is not needed. A gauge can calculate average values, maximums, and minimums per transaction, and other statistical calculations.
- **GAUGE64**
- **GAUGEDIV**
- **ID32, ID64** use the numeric ID simply as an identifier but not as a measurement value, such as an error code or an employee ID. No calculations can be performed on the numeric ID.
- **SHORTSTR** use the string ID as an identifier. No calculations can be performed on the string ID.
- **LONGSTR**
Restriction: METRDEF7= can only equal LONGSTR and can be a long string of 32 bytes. METRDEF1–6 cannot equal LONGSTR.

Note: 32 and 64 signify the number of bits in the counter, gauge, divisor, or ID. △

The METRVAL1–7= sends the value of the user-defined metric to the ARM agent for logging when used in the %ARMSTRT, %ARMUPDT, and %ARMSTOP. The value of the user-defined metric must conform to the corresponding user metrics defined in the %ARMGTID. The following example shows the user metrics:

```sas
%let _armacro=1;

%arminit(appname='Sales App', appuser='userxyz');
/* name and define the user defined metrics */
%armgtid(txnname='Sales Order', txndet='Sales Order Transaction',
    metrnam1=aname, metrdef1=count32);
/* aname is the NAME of the metric and can be anything up to 8 characters */
/* start of user defined metric */
/* initial value of the metric */
%armstrt(metrval1=0);

data myfile;
  . /*some SAS statements*/
  .
  end=EOF;
run;

/* value of the metric is the automatic observation */
%armupdt(data='Sales transaction still running...',maconly=no,
    metrvall=_N_);

data myfile;
  . /*more SAS statements*/
  .
  if EOF then
    /* value of the metric is at the highest observation count */
    %armstop(status=0, metrvall=_N_,maconly=no);
  run;
%armend;
```

### Defining Correlators in ARM Macros

A primary or parent transaction can contain several component or child transactions nested within them. Child transactions can also contain other child transactions. It can be very useful to know how much each child transaction contributes to the total response time of the parent transaction. If a failure occurs within a parent transaction, knowing which child transaction contains the failure is also useful information. Correlators are used to track these parent and child transactions.
The use of correlators requires that multiple transaction start handles be active simultaneously. This requires the use of the CORR= and SHDLVAR= options in the %ARMSTRT macro. You define each parent and child transaction in the %ARMSTRT macro using the SHDLVAR= option. For each child transaction, you must also define the parent transaction using the PARNTVAR= option.

Each child transaction is started after the parent transaction starts. Parent or child transactions can be of the same or different transaction classes. You define the transaction classes in the %ARMGTID macro using the TXNIDVAR= option.

Both parent and child transactions can have updates specified in the %ARMUPDT macro. User metrics can be specified for both transaction types in the %ARMSTRT macro if the user metrics were defined in the corresponding transaction class in the %ARMGTID macro.

All child transactions must stop in the %ARMSTOP macro before the parent transaction stops. The sibling (multiple child) transactions can be stopped in any order.

For example, the parent transaction 100 consists of child transactions 110, 120, and 130, each performing a different part of the unit of work represented by the parent transaction 100. The child transaction 120 contains child transactions 121 and 122. Transaction 200 has no child transactions. Here is a code fragment used to create these relationships:

```sas
%arminit(appname='Application",appidvar=appid);
%armgtid(appidvar=appid,txnname='TranCls',txndet='Transaction Class Def', txnidvar=txnid);
%armstrt(txnidvar=txnid,corr=1,shdlvar=HDL100);
%armstrt(txnidvar=txnid,corr=0,shdlvar=HDL200<,...user metrics>);
%armstrt(txnidvar=txnid,corr=2,shldvar=HDL110,parntvar=HDL100);
%armstrt(txnidvar=txnid,corr=3,shldvar=HDL120,parntvar=HDL100);
%armstrt(txnidvar=txnid,corr=2,shldvar=HDL130,parntvar=HDL100);
%armstrt(txnidvar=txnid,corr=2,shldvar=HDL121,parntvar=HDL120);
%armstrt(txnidvar=txnid,corr=2,shldvar=HDL122,parntvar=HDL120);
...
%armstop(shdlvar=HDL200);
%armstop(shdlvar=HDL121);
%armstop(shdlvar=HDL122);
%armstop(shdlvar=HDL120);
%armstop(shdlvar=HDL130);
%armstop(shdlvar=HDL110);
%armstop(shdlvar=HDL100);
```

### Enabling ARM Macro Execution

#### Setting the _ARMEXEC Macro Variable

All ARM macros are disabled by default so that insertion of ARM macros within created code will not result in inadvertent, unwanted logging. To globally activate execution of the ARM macros, you must set the _ARMEXEC global macro variable to a value of 1. Any other value for _ARMEXEC disables the ARM macros.

There are two methods of setting the _ARMEXEC macro variable. The first method sets the variable during DATA step or SCL program compilation using %LET:

```sas
%let _armexec = 1;
```
If the _ARMEXEC value is not set to 1, then no code is generated and a message is written in the log:

**NOTE:** ARMSTRT macro bypassed by _armexec.

The second method of setting _ARMEXEC variables is to use SYMPUT during execution. To set the _ARMEXEC variable during DATA step or SCL program execution:

```sas
call symput('_armexec', '1');
```

With this technique, the macro checks the _ARMEXEC variable during program execution and the ARM function call is executed or bypassed as appropriate.

---

## Enabling ARM Macro Execution with SCL

There are two methods of setting the _ARMEXEC macro variable—during compilation or execution. Both methods are explained in “Setting the _ARMEXEC Macro Variable” on page 1063, or you can use a combination of these methods. For example, set _ARMEXEC to 1 using the compilation technique (perhaps in an autoexec at SAS initialization), and then code a drop-down menu option or other means within the application to turn _ARMEXEC on and off dynamically using CALL SYMPUT.

In SCL, if _ARMEXEC is not 1, when the program compiles, all macros will be set to null and the ARM interface will be unavailable until it is recompiled with _ARMEXEC set to 1.

Additionally, to enable proper compilation of the ARM macros within SCL, you must set the _ARMSCL global macro variable to 1 prior to issuing any ARM macros. This variable suppresses the generation of DROP statements, which are invalid in SCL.

---

## Conditional ARM Macro Execution

It is useful to code the ARM macros in your program but to execute them only when needed. All ARM macros support a LEVEL= option that specifies the execution level of that particular macro.

If it is coded, the execution level of the macro is compared to two global macro variables, _ARMGLVL and _ARMTLVL. _ARMGLVL is the global level macro variable. If the LEVEL= value on the ARM macro is less than or equal to the _ARMGLVL value, then the macro is executed. If the LEVEL= value on the ARM macro is greater than the _ARMGLVL value, then macro execution is bypassed:

```sas
/* Set the global level to 10 */
%let _armglvl = 10;

data _null_;  
  %arminit(appname='App 1', appuser='userid' );
  %armgtid(txnname='Txn 1', txndet='Transaction #1 detail' );

  /* These macros are executed */
  %armstrt( level=9 );
  %armstop( level=9 );

  /* These macros are executed */
  %armstrt( level=10 );
  %armstop( level=10 );

  /* These macros are NOT executed */
  %armstrt( level=11 );
```
%armstop( level=11 );

%armend
run;

_ARMTLVL is the target level macro variable and works similarly to the ARMGLVL, except the LEVEL= value on the ARM macro must be exactly equal to the _ARMTLVL value for the macro to execute:

/* Set the target level to 10 */
%let _armlvl = 10;

data _null_;  
%arminit(appname='Appl 1', appuser='userid');
%armgtid(txnname='Txn 1', txndet='Transaction #1 detail');

/* These macros are NOT executed */
%armstrt( level=9 );
%armstop( level=9 );

/* These macros are executed */
%armstrt( level=10 );
%armstop( level=10 );

/* These macros are NOT executed */
%armstrt( level=11 );
%armstop( level=11 );

%armend
run;

The LEVEL= option can be placed on any ARM macro and this is highly recommended. It allows you to design more granular levels of logging that can serve as an effective filtering device by logging only as much data as you want. If you set both _ARMGLVL and _ARMTLVL at the same time, then both values are compared to determine whether the macro should be executed or not.

---

## Setting the Macro Environment

You set the global ARM macro environment by using the _ARMACRO variable with the value of 1 or 0. The value of 1 specifies that all ARM macros occur in open code and the value of 0 specifies that the ARM macros occur only in DATA steps. You use the MACONLY= option if an individual ARM macro is not placed outside of the global environment that is defined by the _ARMACRO setting. For SCL programs, you specify _ARMSCL with a value of 1.

The following table shows the global value, the temporary option needed, and the results.
Table 6.3 Using _ARMACRO and _ARMSCL to Set the ARM Macro Environment

<table>
<thead>
<tr>
<th>Global Value</th>
<th>Temporary Option</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>%let _ARMACRO=0;</td>
<td>MACONLY=NO</td>
<td>macro is in DATA step</td>
</tr>
<tr>
<td>%let _ARMACRO=1;</td>
<td>MACONLY=YES</td>
<td>macro is in open code</td>
</tr>
<tr>
<td>%let _ARMSCL=1;</td>
<td>none</td>
<td>macro is in SCL</td>
</tr>
<tr>
<td>%let _ARMSCL=0;</td>
<td>none</td>
<td>macro is not in SCL</td>
</tr>
</tbody>
</table>

The following example shows how to set the macro environment in a DATA step:

```sas
/* set global environment */
%let _armacro = 1;

data _null_;  
  %arminit(appname='Appl 1', appuser='user1', appidvar=appl, maconly=no);
  /* exception to global value */
run;

/* using global setting */
/* maconly= parameter not needed */
%armgtid(txnname='Txn 1A', txndet='Txn Class 1A', appidvar=appl, txnidvar=txnla);
```

The following example shows how to set the macro environment in SCL using autoexec:

```sas
/* set global environment */
%let _armscl = 1;
%let _armexec = 1;

The following example shows how to set the macro environment in SCL:

```sas
init:
  %arminit(appname='Appl 1', appuser='user1', appidvar=appl);
  %armgtid(txnname='Txn 1A', txndet='Txn Class 1A', appidvar=appl, txnidvar=txnla);
return;
main:
  %armstrt(txnidvar=txn1a, shdlvar=strt1);
return;
term:
  %armstop(shdlvar=strt1);
  %armend(appidvar=app1);
return;
```

Using ARM Post-Processing Macros

Post-processing ARM macros are also available. These ARM macros are specific to the SAS ARM implementation; they are not part of the ARM API standard.
After logging performance data to the ARM log, you can then use the ARM macros to read the log and create SAS data sets for reporting and analysis. The default name of the file is ARMLOG.LOG. To specify a different output file, use the ARMLOC= system option. There are three ARM post-processing macros:

- **%ARMCONV** converts an ARM log created in SAS 9 or later, which uses a simple format, into the label=item ARM format used in SAS 8.2.

- **%ARMPROC** processes the ARM log and outputs six SAS data sets that contain the information from the log. It processes ARM logs that include user metadata definitions on class transactions and user data values on start handles, update, and stop transactions.

- **%ARMJOIN** processes the six SAS data sets that are created by %ARMPROC and creates data sets and SQL views that contain common information about applications and transactions.

For SAS 9 or later, the simple format of the ARM log records is comma delimited, which consists of columns of data separated by commas. The datetime stamp and the call identifier always appear in the same column location.

The format used in SAS 8.2 is a label=item format and is easier to read. Unfortunately, SAS spends a lot of time formatting the string, and this affects performance. In SAS 9 or later, ARM logs can be compared with SAS 8.2 ARM logs by using the following methods:

1. You can generate the SAS 9 or later ARM log and process it using the SAS 9 or later %ARMPROC and %ARMJOIN. The resulting data sets contain the complete statistics for the end of the start handle and the end of the application.

2. To convert SAS 9 or later ARM log format to the SAS 8.2 format you use the SAS ARM macro %ARMCONV.

## Troubleshooting Error Messages

This section provides information on possible causes to error messages that you may encounter when using the ARM macros. The following table lists the error messages and possible causes:

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Possible Causes</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ARMINIT: Negative app ID returned</td>
<td>□ Invalid appidvar variable type.</td>
</tr>
<tr>
<td></td>
<td>□ Error in SAS ARM environment.</td>
</tr>
<tr>
<td>%ARMGTID: Negative txn ID returned</td>
<td>□ Invalid appidvar or appidvar value or variable type.</td>
</tr>
<tr>
<td></td>
<td>□ Valid appidvar value, ARM application has been closed.</td>
</tr>
<tr>
<td>%ARMSTRT: Invalid optional data parameter(s)</td>
<td>□ Missing parentheses when correlator requested on previous %ARM call.</td>
</tr>
<tr>
<td>- check syntax.</td>
<td></td>
</tr>
<tr>
<td>Category SAS ARM Macro</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ARM Macro</td>
<td>“%ARMEND Macro” on page 1070</td>
</tr>
<tr>
<td></td>
<td>“%ARMGTID Macro” on page 1072</td>
</tr>
<tr>
<td></td>
<td>“%ARMINIT Macro” on page 1074</td>
</tr>
<tr>
<td></td>
<td>“%ARMSTOP Macro” on page 1079</td>
</tr>
<tr>
<td></td>
<td>“%ARMSTRT Macro” on page 1081</td>
</tr>
<tr>
<td></td>
<td>“%ARMUPDT Macro” on page 1084</td>
</tr>
<tr>
<td></td>
<td>Signifies the termination of an application</td>
</tr>
<tr>
<td></td>
<td>Assigns a unique identifier to a transaction class</td>
</tr>
<tr>
<td></td>
<td>Signifies the initialization of an application</td>
</tr>
<tr>
<td></td>
<td>Marks the end of a transaction instance</td>
</tr>
<tr>
<td></td>
<td>Signals the start of execution of a transaction instance</td>
</tr>
<tr>
<td></td>
<td>and returns a unique handle that is passed to %ARMUPDT and %ARMSTOP</td>
</tr>
<tr>
<td></td>
<td>Updates a transaction that has been previously started</td>
</tr>
</tbody>
</table>
%ARMCONV Macro

Converts a SAS System 9 or later ARM log written in simple format to the more readable label=item ARM format used in Release 8.2.

Category: ARM Post-Processing Macro

Syntax

%ARMCONV(login='aname',logout='aname');

Required Arguments

LOGIN='aname'

is the physical name of the input ARM log.

LOGOUT='aname'

is the physical name of the output ARM log.

Details

There are two ARM formats:

- the SAS System 9 simple format, which is comma delimited
- a label=item format that was created in Release 8.2.

The simple format for ARM log records minimizes the resources it takes to log ARM records. It does minimal formatting and delimits each item with a comma. The label=item format is the format used in Release 8.2. The label=item format requires time to format the string and decreases performance. SAS System 9 or later writes all ARM log records in simple format only. The %ARMCONV macro is used to convert a SAS System 9 or later ARM log to the label=item format.
Examples

Example 1: SAS System 9 ARM Log

I,1326895477.699000,1,1.792577,1.592289,DATASTEP;WIN_NT;9.00.00A0D011602,*
G,1326895478.60000,1,1,putc,
I,1326895478.110000,2,2.153096,1.632347,DATASTEP;WIN_NT;9.00.00A0D011602;ABS,*
G,1326895478.200000,2,2,putc,
I,1326895478.240000,3,2.283283,1.632347,DATASTEP;WIN_NT;9.00.00A0D011602;REL,*
G,1326895478.340000,3,3,putc,
G,1326895478.861000,1,4,,$,PUTC
S,1326895479.532000,1,4,1,3.434939,1.752520
P,1326895489.166000,1,4,1,12.938604,1.792577,0
G,1326895489.296000,1,5,$20.,PUTC

The following code converts the SAS System 9 or later ARM log to a Release 8.2 ARM log:

```plaintext
%armconv(login='u:\arm\putc.log',logout='f:\arm\armlogs\putcV8.log');
run;
```

Example 2: Release 8.2 ARM Log

17JAN2002:14:04:37.699 ARM_INIT AppID=1AppName=DATASTEP;WIN_NT;9.00.00A0D011602 AppUser=*
17JAN2002:14:04:38.600 ARM_GETID AppID=1 ClsID=1 TxName=putc TxDet=
17JAN2002:14:04:38.110 ARM_INIT AppID=2 AppName=DATASTEP;WIN_NT;9.00.00A0D011602;ABS AppUser=*
17JAN2002:14:04:38.200 ARM_GETID AppID=2 ClsID=2 TxName=putc TxDet=
17JAN2002:14:04:38.240 ARM_INIT AppID=3 AppName=DATASTEP;WIN_NT;9.00.00A0D011602;REL AppUser=*
17JAN2002:14:04:38.340 ARM_GETID AppID=3 ClsID=3 TxName=putc TxDet=
17JAN2002:14:04:38.861 ARM_GETID AppID=1 ClsID=4 TxName=$ TxDet=PUTC
17JAN2002:14:04:39.532 ARM_START AppID=1 ClsID=4 TxSHdl=1 TxName=$ TxDet=PUTC
17JAN2002:14:04:49.166 ARM_STOP AppID=1 ClsID=4 TxSHdl=1 TxStat=0 TxElap=0:00:12.939
TxCpu=0:00:01.793
17JAN2002:14:04:49.296 ARM_GETID AppID=1 ClsID=5 TxName=$20. TxDet=PUTC

%ARMEND Macro

Signifies the termination of an application

Category: ARM Macro

Syntax

```plaintext
%ARMEND (options);
```
Options

**APPID=SAS numeric variable or constant**

is the application ID to use on the ARM_GETID function call. The value must be a SAS numeric variable or constant.

*Note:* Use APPIDVAR= instead of APPID= when coding new applications. APPID= is obsolete.

**APPIDVAR=SAS numeric variable**

is a SAS numeric variable that supplies the value of the application ID.

**LEVEL= variable**

is a variable that specifies the conditional execution level. The value must be a numeric constant or variable.

**MACONLY=NO | YES**

allows the %ARMEND macro to be issued in open code, outside of a DATA step. You set the value to YES if the macros are in open code and NO if they are in a DATA step.

*Default:* NO

**SCL=NO | YES**

is used only in SCL programs and specifies whether the macro is in an SCL environment. Valid values are YES and NO (no quotation marks).

*Default:* NO

Details

Use %ARMEND when you are finished initiating new activity using the ARM API. %ARMEND is typically called when an application or user instance is terminating. Each %ARMEND is paired with one %ARMINIT to mark the end of an application. %ARMEND is a signal from the application that it will not issue any more ARM calls, and it is typically executed at application termination. ARM calls issued after an application has been ended with %ARMEND result in an error.

*Note:* You must terminate ARM with %ARMEND to avoid getting a warning or an error from %ARMPROC.

Input

The input is an application ID that is generated by a previous %ARMINIT macro. If the APPID= or APPIDVAR= options are supplied, the specified value is used as the application ID. Otherwise, the value of the global macro variable _ARMAPID is used.

Output

The _ARMRC variable, which is the error status code that was returned from the ARM_END function call, is the output.
Examples

Example 1: Basic Usage

data _null_
   %armend;
run;

Example 2: Supplying an Application ID Using APPIDVAR=

data _null_
   %arminit(appname=aname, appuser='sasxyz', appidvar=myapp);
run;

data _null_
   %armend(appidvar=myapp);
run;

%ARMGTID Macro

Assigns a unique identifier to a transaction class

Category: ARM Macro

Syntax

%ARMGTID(TXNNAME='aname'<,options>);

Required Argument

TXNNAME='aname'
   is a required transaction name that is a SAS character variable or quoted literal value.

   Restriction: The transaction name has a 127–character limit.

Options

APPID=SAS numeric variable or constant
   is an application ID to use on the ARM_GETID function call. It must be a SAS
   numeric variable or constant.

   Note: Use APPIDVAR= instead of APPID= when coding new applications. APPID=
   is obsolete. △

APPIDVAR=SAS numeric variable
   is a SAS numeric variable that supplies the value of the application ID.

LEVEL=numeric constant or variable
   is a variable that specifies the conditional execution level. The value must be a
   numeric constant or variable.
MACONLY=NO | YES
allows the %ARMINIT macro to be issued in open code, outside of a DATA step. You set the value to YES if the macros are in open code and NO if they are in a DATA step.
Default: NO

METRNAM1–7=’aname’
is the user name for the user-defined metric and must be a SAS character variable or quoted literal value.
Requirement: The user name and user-defined metric definition must be specified together.

METRDEF1–7=option
is the definition of the user-defined metric. The value must be one of the following:
COUNT32,
COUNT64,
COUNTDIV
use the counter to sum up the values over an interval. A counter can also calculate average values, maximums, and minimums per transaction, and other statistical calculations.
GAUGE32,
GAUGE64,
GAUGEDIV
use the gauge when a sum of values is not needed. A gauge can calculate average values, maximums, and minimums per transaction, and other statistical calculations.
ID32, ID64
use the numeric ID simply as an identifier but not as a measurement value, such as an error code or an employee ID. No calculations can be performed on the numeric ID.
SHORTSTR,
LONGSTR
use the string ID as an identifier. No calculations can be performed on the string ID.
Restriction: METRDEF7 can only equal LONGSTR. METRDEF1–6 cannot equal LONGSTR.
Requirement: The user name and user-defined metric definition must be specified together.

SCL=NO | YES
is used only in SCL programs and specifies whether the macro is in an SCL environment. Valid values are YES and NO (no quotation marks).
Default: NO

TXNDET=’aname’
is a transaction detail that is a SAS character variable or quoted literal value.
Restriction: The transaction detail has a 127–character limit.

TXNIDVAR=SAS numeric variable
is a SAS numeric variable that contains the value of the transaction ID.

Details
%ARMGTID is used to name a transaction class. Transaction classes are related units of work within an application. One or more %ARMGTID calls are typically issued when the application starts in order to define each of the transaction classes used by the application.

Input
The input is an application ID generated by a previous %ARMINIT macro. If the APPID= or APPIDVAR= options are supplied, the supplied value is used as the application ID. Otherwise, the value of the global macro variable _ARMAPID ID used.
Output

The _ARMTXID variable, which is the transaction class ID that was returned from the ARM_GETID function call, is the output. Any variable that the TXNIDVAR= option points to is also updated.

Examples

Example 1: Basic Usage

data _null_;  
%armgtid(txnname='txn OE', txndet='Order Entry txn class');  
run;

Example 2: Saving the Transaction ID

data _null_;  
%arminit(appname=aname, appuser='sasxyz');  
%armgtid(txnname='txn OE', txndet='Order Entry txn class',  
        txnidvar=txn1);  
put 'transaction id is ' txn1;  
run;

%ARMINIT Macro

Signifies the initialization of an application

Category: ARM Macro

Syntax

%ARMINIT(APPNAME='aname'<,options>);

Required Argument

APPNAME='aname'

is the required application name that is a SAS character variable or quoted literal value.

Restriction: The application name has a 127–character limit.

Options

APPIDVAR=SAS numeric variable

is a SAS numeric variable containing the value of the application ID.

APPUSER='aname'

is the application user ID that is a SAS character variable or quoted literal value.

Restriction: The application user ID has a 127–character limit.
GETID=NO | YES
is optional and denotes whether to generate an ARM_GETID function call after the ARM_INIT. If the GETID value is YES, you can define the user metrics.
Default: NO
Requirement: TXNNAME= is required when you use GETID=YES.

LEVEL=numeric constant or variable
is a variable that specifies the conditional execution level. The value must be a numeric constant or variable.

MACONLY=NO | YES
allows the %ARMINIT macro to be issued in open code, outside of a DATA step. You use YES if the macros are in open code and NO if they are in a DATA step.
Default: NO

SCL=NO | YES
is used only in SCL programs and specifies whether the macro is in an SCL environment. Valid values are YES and NO (no quotation marks).
Default: NO

TXNIDVAR=SAS numeric variable
is a SAS numeric variable that contains the value of a transaction ID.
Restriction: Use the TXNIDVAR= only when using GETID=YES.

TXNDET='aname'
is a transaction detail that is a SAS character variable or quoted literal value and can be specified only when GETID=YES.
Restriction: The transaction detail has a 127-character limit.

TXNNAME='aname'
is a transaction name that is a SAS character variable or quoted literal value.
Requirement: TXNNAME= is required only when using GETID=YES.

Details
A %ARMINIT macro call names the application and optionally the user of the application. Additionally, it initializes the ARM environment if a previous %ARMINIT has not been issued. It typically is executed when the application initializes.

Input
None.

Output
The _ARMAPID variable, which is the application ID that was returned from the ARM_INIT function call, is the output. If GETID=YES, then the _ARMTXID is returned also. Any variables for the APPIDVAR= and TXNIDVAR= are also updated.

Examples

Example 1: Basic Usage
```sas
data _null_
  %arminit(appname='General Ledger');
run
```
Example 2: Supplying the User ID

data _null_
 aname = 'Order Entry Application';
%arminit(appname=aname, appuser='sasxyz');
run;

Example 3: Generating an ARM_GETID in Addition to an ARM_INIT

data _null_
  %arminit(appname='Warehouse App', getid=YES,
              txnname='Query 1', txndet='My long query');
run;

Example 4: Saving the Application ID

data _null_
  %arminit(appname=aname, appuser='sasxyz', appidvar=appl);
  put 'application id is ' appl;
run;

%ARMJOIN Macro

Reads the six SAS data sets created by the %ARMPROC macro and creates SAS data sets and SQL views that contain common information about applications and transactions.

Category: ARM Post-Processing Macro

Syntax

%ARMJOIN(<option(s)>);

Options

LIBIN=libref
is the libref for the SAS data library that contains the six data sets created by %ARMPROC.
Default: WORK

LIBOUT=libref
is the libref for the SAS data library that contains the application and transaction data sets.
Default: WORK

TXNDS=YES | NO
specifies whether the transaction data sets are to be created.
Default: YES

UPDTDS=YES | NO
specifies whether the update data sets are to be created.
Default: YES
Details

The %ARMJOIN macro reads the six output data sets produced by %ARMPROC. It merges the information from those data sets to produce a variety of output data sets and views for easier reporting of ARM data.

Note: The %ARMJOIN macro does not work from SCL. It must be run in the DATA step environment. △

Input

The input is the SAS data sets from %ARMPROC; therefore, you must run %ARMPROC before running %ARMJOIN.

Output

The output is a single SAS library containing
- information about applications (APP)
- a DATA step view that contains information about all start handles, including parent correlator class and parent start handles (TXNVIEW)
- a view that contains information about all update transactions (UPDTVIEW)
- one transaction data set for each application
- one update data set for each application.

The application data set is named APP and contains one observation for every application that is found in the input data. Each observation contains information such as application name, user ID, transaction counts, average application response time, and so on. Additionally, each observation contains a numeric variable “appno” that is the identifier of the related transaction or update data set that contains more detailed transaction information.

The transaction data sets are named TXN1, TXN2, TXN3, and so on. Each data set corresponds to a single application and each observation represents a single ARM transaction containing start and stop times, elapsed times, and CPU time.

The TXNVIEW view joins all transaction data sets and presents them as a single data set. Start handle elapsed time and CPU time are calculated from the start and stop transactions. If the start handle has a parent start handle, the class ID and start handle of the parent are included using the variables PARCLS= and PARHDL=. If no parent is specified, these variables will contain missing values.

The update data sets are named UPDT1, UPDT2, UPDT3, and so on. Each data set corresponds to a single application and contains multiple observations for each ARM transaction. Each observation contains the ARM call datetime, an ARM call sequence ID, and if applicable, elapsed time, CPU time, and update data.

The UPDTVIEW view joins all update data sets and presents them as a single data set.

The transaction data sets are easier to use for analyzing individual ARM transactions because all information about a transaction is collapsed into one observation. However, the transaction data sets do not contain any information from %ARMUPDT calls.

The update data sets are similar to the transaction data set; however, information about a single transaction is spread over several observations. These data sets contain logged data buffer information from all %ARMUPDT calls.
Examples

Example 1: Basic Usage

```sas
filename ARMLOG 'd:\armlog';
%armproc();
%armjoin();
```

Example 2: Defining a Permanent Library to Read %ARMPROC Output and Store %ARMJOIN Views

```sas
libname user 'c:\arm\user';
%armjoin(libin=user,libout=user);
run;
```

%ARMPROC Macro

Processes an input ARM log and outputs six SAS data sets that contain the gathered information from the log

**Category:** ARM Post-Processing Macro

**Syntax**

```
%ARMPROC(<option(s)>);
```

**Options**

- **LIB=libref**  
  is the libref for the SAS data library to contain the six data sets.  
  **Default:** WORK

- **LOG=pathname**  
  is the pathname for the physical location of the ARM log. If a pathname is not supplied, you must preassign the ARMLOG fileref before calling the macro.

- **LOGNEW=pathname**  
  is the pathname the physical location of the new ARM log uses when ARM processing is resumed.

**Details**

The %ARMPROC macro reads an ARM log and outputs six SAS data sets that contain the information from the log. This macro reads the variable name and value pairs from the ARM log as named input (var=value). You should either preassign the ARMLOG fileref prior to calling the macro or supply the LOG= option. If the ARMLOC= option is ignored, an actual FILENAME statement is required to preassign the ARMLOG fileref.

**Note:** The %ARMPROC macro does not work from SCL. Any commas that are part of the name cause the log to be parsed incorrectly. Commas in the data area of the UPDATE record do not cause any issues. △
Input
The external file containing the ARM log is the input.

Output
The %ARMPROC macro creates six SAS data sets. These SAS data sets contain information from calls to the ARM API function calls. The six SAS data sets are:

- INIT—contains information from all arm_init calls
- GETID—contains information from all arm_getid calls
- START—contains information from all arm_start calls
- UPDATE—contains information from all arm_update calls
- STOP—contains information from all arm_stop calls
- END—contains information from all arm_end calls.

Examples

Example 1: Defining a Permanent Library to Store %ARMPROC Output
```sas
libname user 'f:\arm\user';
%armproc(lib=user);
run;
```

Example 2: Supplying the LIB= and LOG= Options
```sas
libname armout 'sas library name';
%armproc(lib=armout,log=c:userid\arm\armlog);
```

%ARMSTOP Macro
Marks the end of a transaction instance
Category: ARM Macro

Syntax
```
%ARMSTOP(options);
```

Options

LEVEL=numeric constant or variable
is a variable that specifies the conditional execution level. The value must be a numeric constant or variable.

MACONLY=NO | YES
allows the %ARMSTOP macro to be issued in open code, outside of a DATA step. You set the value to YES if the macros are in open code and NO if they are in a DATA step.

Default: NO
%ARMSTOP Macro

Chapter 6

METRVAL1-7='aname'
is the value of the user-defined metric. The value must be a SAS character variable or a quoted literal value up to eight characters in length.

**Requirement:** The value of the user-defined metric must correspond to the user metrics defined in %ARMGTID.

SCL=NO | YES
is used only in SCL programs and specifies whether the macro is in an SCL environment. Valid values are YES and NO (no quotation marks).

**Default:** NO

SHANDLE=SAS numeric variable or constant
is a start handle to use on the ARM_UPDATE function call. The value must be a SAS numeric variable or constant.

SHDLVAR=SAS numeric variable
is a SAS numeric variable that contains the value of the start handle.

STATUS=SAS numeric variable or numeric constants
is a transaction status value to pass to the ARM_STOP function call. The value must be a SAS numeric variable or numeric constants 0, 1, or 2. The default is 0.

**Details**

%ARMSTOP signals the end of a transaction, that was started using an %ARMSTRT macro call.

**Input**
The input is a start handle that is generated from a previous %ARMSTRT call. If the SHANDLE= or SHDLVAR= options are supplied, the specified value is used as the start handle. Otherwise, the value of the global macro variable _ARMSHDL is used.

**Output**
The _ARMRC variable, which contains the error status code that was returned from the ARM_STOP function call, is the output.

**Examples**

**Example 1: Basic Usage**
```sas
data _null_;  
  %armstop; /* status will default to zero*/  
run;
```

**Example 2: Supplying a Non-Zero Status**
```sas
data _null_;  
  rc = 2;  
  %armstop(status=rc);  
run;
```

**Example 3: Supplying a Start Handle Using SHDLVAR=**
```sas
data _null_;  
  %arminit(appname=aname, appuser='sasxyz');
```
%ARMSTRT Macro

Signals the start of execution of a transaction instance and returns a unique handle that is passed to %ARMUPDT and %ARMSTOP

Category: ARM Macro

Syntax

%ARMSTRT(options);

Options

APPID=SAS numeric variable or constant
is an application ID to use on the ARM_GETID function call. The value must be a SAS numeric variable or constant.

Restriction: Use the APPID= only when using GETID=YES. See %ARMINIT for information about GETID=YES.

Note: Use APPIDVAR= instead of APPID= when coding new applications. APPID= is obsolete. △

APPIDVAR=SAS numeric variable
is a SAS numeric variable that supplies the value of the application ID.

Restriction: Use the APPIDVAR= only when using GETID=YES. See %ARMINIT for information about GETID=YES.

CORR=n
is an option to define the type of parent and child transactions.

Default: 0

Requirement: You use CORR= only when using correlators.

GETID=NO | YES
is optional and denotes whether to generate an ARM_GETID before the ARM_START. If the GETID value is YES, you can define the user metrics.

Requirement: TXNNAME= is required when using GETID=YES.

Default: NO

LEVEL=numeric constant or variable
is a variable that specifies the conditional execution level. The value must be a numeric constant or variable.
MACONLY=NO | YES
allows the %ARMSTRT macro to be issued in open code, outside of a DATA step. You
set the value to YES if the macros are in open code and NO if they are in a DATA
step.
 Default: NO

METRVAL1–7=’aname’
is the value of the user-defined metric. The value must be a SAS character variable
or a quoted literal value up to eight characters in length.
Requirement: The value of the user-defined metric must correspond to the user
metrics defined in %ARMGTID.

PARNTVAR=SAS numeric variable
is the name of a SAS numeric variable that contains the value of the parent
transaction start handle and is only used when defining a child transaction.
PARNTVAR= is used only when the CORR= option has a value of 2 or 3.

SCL=NO | YES
is used only in SCL programs and specifies whether the macro is in an SCL
environment. Valid values are YES and NO (no quotation marks).
 Default: NO

SHDLVAR=SAS numeric variable
is a SAS numeric variable that contains the value of the start handle. SHDLVAR= is
required when using correlators to define parent and child transactions.

TXNDET=’aname’
is a transaction detail that is a SAS character variable or quoted literal value.
Requirement: TXNDET= is used only with GETID=YES.
Restriction: The transaction detail has a 127-character limit.

TXNID=SAS numeric variable or constant
is a transaction ID to use in the ARM_START function call. The value must be a
SAS numeric variable or constant.
 Note: Use TXNIDVAR= instead of TXNID= when coding new applications. TXNID=
is obsolete. △

TXNIDVAR=SAS numeric variable
is a SAS numeric variable that contains the value of the transaction ID when
GETID=NO. It contains the value of the TXN ID when GETID=YES.

TXNNAME=’aname’
is the transaction name that is a SAS character variable or quoted literal value.
Requirement: TXNNAME= is required only when using GETID=YES.
Restriction: The transaction name has a 127-character limit.

Details

%ARMSTRT signals the start of a transaction, also known as a transaction instance.
A transaction instance is an instantiation of a transaction class that was previously
defined by %ARMGTID.
If user metrics are defined for a transaction class using %ARMGTID, the value for
the user metrics begins with the METRVAL1–7= option.
The CORR= option defines the type of parent (primary) and child (component)
transactions using the following values:

0 not part of a related group
%ARMSTRT Macro 1083

1 parent transaction
2 child transaction
3 child of one transaction and parent of another.

Note: You use CORR= only when using correlators.

Each child start handle variable must be accompanied with a parent start handle variable. Below is a code fragment to show the use of correlator types and the SHLDVAR= and PARNTVAR= options:

```sas
%armstrt(txnidvar=txnid,corr=1,shdlvar=HDL100);
%armstrt(txnidvar=txnid,corr=0,shdlvar=HDL200<...user metrics>);
%armstrt(txnidvar=txnid,corr=2,shldvar=HDL110,parntvar=HDL100);
%armstrt(txnidvar=txnid,corr=3,shldvar=HDL120,parntvar=HDL100);
```

**Input**

The transaction class ID that is generated by a previous %ARMGTID call is the input. If the TXNID= or TXNIDVAR= options are supplied, the supplied value is used as the transaction ID. Otherwise, the value of the global macro variable _ARMTXID is used.

If GETID=YES and the APPID= or APPIDVAR= options are supplied, the supplied value is used as the application ID. Otherwise, the value of the global macro variable _ARMAPID is used.

**Output**

The _ARMSHDL variable, which is the start handle that was returned from the ARM_START function call, is the output. If GETID=YES, then the _ARMTXID variable is updated also. The variables that TXNIDVAR= and SHDLVAR= point to are also updated if supplied.

**Examples**

**Example 1: Basic Usage**

```sas
data _null_
  %arminit(appname='Forecast')
  %armgtid(txnname='Txn 1A', txndet='Forecasting Txn Class');
  %armstrt;
run;
```

**Example 2: Supplying the Transaction ID Using TXNIDVAR=**

```sas
data _null_
  %arminit(appname=aname, appuser='sasxyz');
  %armgtid(txnname='txn OE', txndet='Order Entry txn class' txnidvar=txnnum);

data _null_
  %armstrt(txnidvar=txnname);
run;
```

**Example 3: Generating an ARM_GETID Call and an ARM_START**

```sas
data _null_
  %arminit(appname='Forecast', appidvar=savapp);
```

run;

data _null_
   %armstrt(getid=YES, txname='Txn 1A',
           txndet='Forecasting Txn Class',
           appidvar=savapp);
run;

%ARMUPDT Macro

Updates a transaction that has been previously started

Category: ARM Macro

Syntax

%ARMUPDT(DATA=<option>, <options>);

Recommended Argument

DATA='variable'

is a SAS character variable or a quoted literal user-supplied data buffer that contains
text to pass to the ARM_UPDATE function call. DATA = is not required but highly
recommended. This information is mutually exclusive of user-defined metric values.

Restriction: The data value has a 1,020-character limit.

Options

LEVEL=numeric constant or variable

is a variable that specifies the conditional execution level. The value must be a
numeric constant or variable.

MACONLY=NO | YES

allows the %ARMUPDT macro to be issued in open code, outside of a DATA step. You
set the value to YES if the macros are in open code and NO if they are in a DATA
step.

Default: NO

METRVAL1–7='aname'

is the value of the user-defined metric. The value must be a SAS character variable
or a quoted literal value up to eight characters in length. These values are ignored if
the DATA= option is used.

Requirement: The value of the user-defined metric must correspond to the user
metrics defined in %ARMGTID.

SCL=NO | YES

is used only in SCL programs and specifies whether the macro is in an SCL
environment. Valid values are YES and NO (no quotation marks).
Default: NO

**SHANDLE=SAS numeric or constant**

is a start handle to use on the ARM_UPDATE function call. The value is a SAS numeric or constant.

*Note:* Use SHDLVAR= instead of SHANDLE= when coding new applications. SHANDLE= is obsolete.

**SHDLVAR=SAS numeric variable**

is a SAS numeric variable that contains the value of the start handle.

**Details**

%ARMUPDT is a call that you can execute any number of times after a %ARMSTRT and before a %ARMSTOP. It allows you to supply any additional information about the transaction in progress.

**Input**

The input is a start handle that is generated by a previous %ARMSTRT call. If the SHANDLE= or SHDLVAR= options are supplied, the specified value is used as the start handle. Otherwise, the value of the global macro variable _ARMSHDL is used.

*Note:* User metric values and user-supplied data buffers are mutually exclusive parameters. Each requires its own update call to get both types of data into the UPDATE records.

**Output**

The _ARMRC variable, which contains the error status code that was returned from the ARM_UPDATE function call, is the output.

**Examples**

**Example 1: Basic Usage**

```sas
data _null_; updtdata = 'Txn still running at' || put(time(), time.); %armupdt(data=updtdata); run;
```

**Example 2: Supplying a Start Handle Using SHDLVAR=**

```sas
data _null_; %arminit(appname=aname, appuser='sasxyz'); %armgtid(txnname='txn OE', txndet='Order Entry txn class'); %armstrt(shdlvar=sh1); run;

data _null_; %armupdt(data='OE txn pre-processing complete', shdlvar=sh1 ); run;
```
CHAPTER 7

Statements

Definition of Statements 1089
DATA Step Statements 1090
   Executable and Declarative Statements 1090
   DATA Step Statements by Category 1091
Global Statements 1095
   Definition 1095
   Global Statements by Category 1096
Dictionary 1100
   ABORT Statement 1100
   ARRAY Statement 1102
   Array Reference Statement 1107
   Assignment Statement 1110
   ATTRIB Statement 1111
   BY Statement 1114
   CALL Statement 1119
   CARDS Statement 1120
   CARDS4 Statement 1120
   CATNAME Statement 1120
   Comment Statement 1123
   CONTINUE Statement 1125
   DATA Statement 1126
   DATALINES Statement 1133
   DATALINES4 Statement 1134
   DECLARE Statement 1135
   DELETE Statement 1140
   DESCRIBE Statement 1141
   DISPLAY Statement 1141
   DM Statement 1143
   DO Statement 1145
   DO Statement, Iterative 1146
   DO UNTIL Statement 1150
   DO WHILE Statement 1151
   DROP Statement 1152
   END Statement 1154
   ENDSAS Statement 1155
   ERROR Statement 1155
   EXECUTE Statement 1156
   FILE Statement 1157
   FILE, ODS Statement 1171
   FILENAME Statement 1172
   FILENAME Statement, CATALOG Access Method 1178
<table>
<thead>
<tr>
<th>Statement/Method</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILENAME, CLIPBOARD Access Method</td>
<td>1181</td>
</tr>
<tr>
<td>FILENAME Statement, EMAIL (SMTP) Access Method</td>
<td>1183</td>
</tr>
<tr>
<td>FILENAME Statement, FTP Access Method</td>
<td>1193</td>
</tr>
<tr>
<td>FILENAME Statement, SOCKET Access Method</td>
<td>1201</td>
</tr>
<tr>
<td>FILENAME Statement, URL Access Method</td>
<td>1205</td>
</tr>
<tr>
<td>FOOTNOTE Statement</td>
<td>1208</td>
</tr>
<tr>
<td>FORMAT Statement</td>
<td>1209</td>
</tr>
<tr>
<td>GO TO Statement</td>
<td>1213</td>
</tr>
<tr>
<td>IF Statement, Subsetting</td>
<td>1214</td>
</tr>
<tr>
<td>IF-THEN/ELSE Statement</td>
<td>1216</td>
</tr>
<tr>
<td>%INCLUDE Statement</td>
<td>1218</td>
</tr>
<tr>
<td>INFILE Statement</td>
<td>1224</td>
</tr>
<tr>
<td>INFORMAT Statement</td>
<td>1244</td>
</tr>
<tr>
<td>INPUT Statement</td>
<td>1248</td>
</tr>
<tr>
<td>INPUT Statement, Column</td>
<td>1263</td>
</tr>
<tr>
<td>INPUT Statement, Formatted</td>
<td>1265</td>
</tr>
<tr>
<td>INPUT Statement, List</td>
<td>1269</td>
</tr>
<tr>
<td>INPUT Statement, Named</td>
<td>1275</td>
</tr>
<tr>
<td>KEEP Statement</td>
<td>1278</td>
</tr>
<tr>
<td>LABEL Statement</td>
<td>1280</td>
</tr>
<tr>
<td>Labels, Statement</td>
<td>1281</td>
</tr>
<tr>
<td>LEAVE Statement</td>
<td>1282</td>
</tr>
<tr>
<td>LENGTH Statement</td>
<td>1283</td>
</tr>
<tr>
<td>LIBNAME Statement</td>
<td>1286</td>
</tr>
<tr>
<td>LIBNAME Statement, SAS/ACCESS</td>
<td>1294</td>
</tr>
<tr>
<td>LIBNAME Statement for SAS/CONNECT Remote Library Services</td>
<td>1294</td>
</tr>
<tr>
<td>LIBNAME Statement for the Metadata Engine</td>
<td>1294</td>
</tr>
<tr>
<td>LIBNAME Statement, SASEDOC</td>
<td>1295</td>
</tr>
<tr>
<td>LIBNAME Statement for SAS/CONNECT TCP/IP Pipe</td>
<td>1295</td>
</tr>
<tr>
<td>LIBNAME Statement for SAS/SHARE</td>
<td>1295</td>
</tr>
<tr>
<td>LIBNAME Statement for Scalable Performance Data</td>
<td>1295</td>
</tr>
<tr>
<td>LIBNAME Statement for the XML Engine</td>
<td>1296</td>
</tr>
<tr>
<td>LINK Statement</td>
<td>1296</td>
</tr>
<tr>
<td>LIST Statement</td>
<td>1297</td>
</tr>
<tr>
<td>%LIST Statement</td>
<td>1300</td>
</tr>
<tr>
<td>LOCK Statement</td>
<td>1301</td>
</tr>
<tr>
<td>LOSTCARD Statement</td>
<td>1304</td>
</tr>
<tr>
<td>MERGE Statement</td>
<td>1306</td>
</tr>
<tr>
<td>MISSING Statement</td>
<td>1308</td>
</tr>
<tr>
<td>MODIFY Statement</td>
<td>1310</td>
</tr>
<tr>
<td><em>NEW</em> Statement</td>
<td>1327</td>
</tr>
<tr>
<td>Null Statement</td>
<td>1330</td>
</tr>
<tr>
<td>ODS CHTML Statement</td>
<td>1331</td>
</tr>
<tr>
<td>ODS <em>ALL</em> CLOSE Statement</td>
<td>1332</td>
</tr>
<tr>
<td>ODS CSVALL Statement</td>
<td>1332</td>
</tr>
<tr>
<td>ODS DECIMAL_ALIGN Statement</td>
<td>1332</td>
</tr>
<tr>
<td>ODS DOCBOOK Statement</td>
<td>1332</td>
</tr>
<tr>
<td>ODS DOCUMENT Statement</td>
<td>1333</td>
</tr>
<tr>
<td>ODS EXCLUDE Statement</td>
<td>1333</td>
</tr>
<tr>
<td>ODS HTML Statement</td>
<td>1333</td>
</tr>
<tr>
<td>ODS HTML3 Statement</td>
<td>1334</td>
</tr>
<tr>
<td>ODS HTMLCSS Statement</td>
<td>1334</td>
</tr>
<tr>
<td>ODS IMODE Statement</td>
<td>1334</td>
</tr>
<tr>
<td>ODS LISTING Statement</td>
<td>1334</td>
</tr>
</tbody>
</table>
Definition of Statements

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 documentation covers two kinds of SAS statements:

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


### DATA Step Statements

#### Executable and Declarative Statements

DATA step statements are executable or declarative statements that can appear in the DATA step. 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**

<table>
<thead>
<tr>
<th>Statement</th>
<th>IF, Subsetting</th>
<th>PUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABORT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assignment</td>
<td>IF-THEN/ELSE</td>
<td>PUT, Column</td>
</tr>
<tr>
<td>CALL</td>
<td>INFILE</td>
<td>PUT, Formatted</td>
</tr>
<tr>
<td>CONTINUE</td>
<td>INPUT</td>
<td>PUT, List</td>
</tr>
<tr>
<td>DECLARE</td>
<td>INPUT, Column</td>
<td>PUT, Named</td>
</tr>
<tr>
<td>DELETE</td>
<td>INPUT, Formatted</td>
<td>PUT, ODS</td>
</tr>
<tr>
<td>DESCRIBE</td>
<td>INPUT, List</td>
<td>PUTLOG</td>
</tr>
<tr>
<td>DISPLAY</td>
<td>INPUT, Named</td>
<td>REDIRECT</td>
</tr>
<tr>
<td>DO</td>
<td>LEAVE</td>
<td>REMOVE</td>
</tr>
<tr>
<td>DO, Iterative</td>
<td>LINK</td>
<td>REPLACE</td>
</tr>
<tr>
<td>DO UNTIL</td>
<td>LIST</td>
<td>RETURN</td>
</tr>
<tr>
<td>DO WHILE</td>
<td>LOSTCARD</td>
<td>SELECT</td>
</tr>
<tr>
<td>ERROR</td>
<td>MERGE</td>
<td>SET</td>
</tr>
<tr>
<td>EXECUTE</td>
<td>MODIFY</td>
<td>STOP</td>
</tr>
<tr>
<td>FILE</td>
<td><em>NEW</em></td>
<td>Sum</td>
</tr>
<tr>
<td>FILE, ODS</td>
<td>Null</td>
<td>UPDATE</td>
</tr>
<tr>
<td>GO TO</td>
<td>OUTPUT</td>
<td></td>
</tr>
</tbody>
</table>
In addition to being either executable or declarative, SAS DATA step statements can be grouped into five functional categories:

### Table 7.1 Categories of DATA Step Statements

<table>
<thead>
<tr>
<th>Category</th>
<th>Let you ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>□ create and modify variables</td>
</tr>
<tr>
<td></td>
<td>□ select only certain observations to process in the DATA step</td>
</tr>
<tr>
<td></td>
<td>□ look for errors in the input data</td>
</tr>
<tr>
<td></td>
<td>□ work with observations as they are being created</td>
</tr>
<tr>
<td>Control</td>
<td>□ skip statements for certain observations</td>
</tr>
<tr>
<td></td>
<td>□ change the order that statements are executed</td>
</tr>
<tr>
<td></td>
<td>□ transfer control from one part of a program to another</td>
</tr>
<tr>
<td>File-handling</td>
<td>□ work with files used as input to the data set</td>
</tr>
<tr>
<td></td>
<td>□ work with files to be written by the DATA step</td>
</tr>
<tr>
<td>Information</td>
<td>□ give SAS additional information about the program data vector</td>
</tr>
<tr>
<td></td>
<td>□ give SAS additional information about the data set or data sets that are being created.</td>
</tr>
<tr>
<td>Window Display</td>
<td>□ display and customize windows.</td>
</tr>
</tbody>
</table>

The following table lists and briefly describes the DATA step statements by category.
### Table 7.2 Categories and Descriptions of DATA Step Statements

<table>
<thead>
<tr>
<th>Category</th>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>“ABORT Statement” on page 1100</td>
<td>Stops executing the current DATA step, SAS job, or SAS session</td>
</tr>
<tr>
<td></td>
<td>“Assignment Statement” on page 1110</td>
<td>Evaluates an expression and stores the result in a variable</td>
</tr>
<tr>
<td></td>
<td>“CALL Statement” on page 1119</td>
<td>Invokes or calls a SAS CALL routine</td>
</tr>
<tr>
<td></td>
<td>“DECLARE Statement” on page 1135</td>
<td>Declares a DATA step component object; creates an instance of and initializes data for a DATA step component object</td>
</tr>
<tr>
<td></td>
<td>“DELETE Statement” on page 1140</td>
<td>Stops processing the current observation</td>
</tr>
<tr>
<td></td>
<td>“DESCRIBE Statement” on page 1141</td>
<td>Retrieves source code from a stored compiled DATA step program or a DATA step view</td>
</tr>
<tr>
<td></td>
<td>“ERROR Statement” on page 1155</td>
<td>Sets <em>ERROR</em> to 1 and, optionally, writes a message to the SAS log</td>
</tr>
<tr>
<td></td>
<td>“EXECUTE Statement” on page 1156</td>
<td>Executes a stored compiled DATA step program</td>
</tr>
<tr>
<td></td>
<td>“IF Statement, Subsetting” on page 1214</td>
<td>Continues processing only those observations that meet the condition</td>
</tr>
<tr>
<td></td>
<td>“LIST Statement” on page 1297</td>
<td>Writes to the SAS log the input data record for the observation that is being processed</td>
</tr>
<tr>
<td></td>
<td>“LOSTCARD Statement” on page 1304</td>
<td>Resynchronizes the input data when SAS encounters a missing or invalid record in data that has multiple records per observation</td>
</tr>
<tr>
<td></td>
<td>“<em>NEW</em> Statement” on page 1327</td>
<td>Creates an instance of a DATA step component object</td>
</tr>
<tr>
<td></td>
<td>“Null Statement” on page 1330</td>
<td>Signals the end of data lines; acts as a placeholder</td>
</tr>
<tr>
<td></td>
<td>“OUTPUT Statement” on page 1341</td>
<td>Writes the current observation to a SAS data set</td>
</tr>
<tr>
<td></td>
<td>“PUTLOG Statement” on page 1373</td>
<td>Writes a message to the SAS log</td>
</tr>
<tr>
<td></td>
<td>“REDIRECT Statement” on page 1375</td>
<td>Points to different input or output SAS data sets when you execute a stored program</td>
</tr>
<tr>
<td></td>
<td>“REMOVE Statement” on page 1377</td>
<td>Deletes an observation from a SAS data set</td>
</tr>
<tr>
<td></td>
<td>“REPLACE Statement” on page 1381</td>
<td>Replaces an observation in the same location</td>
</tr>
<tr>
<td></td>
<td>“STOP Statement” on page 1407</td>
<td>Stops execution of the current DATA step</td>
</tr>
<tr>
<td></td>
<td>“Sum Statement” on page 1408</td>
<td>Adds the result of an expression to an accumulator variable</td>
</tr>
<tr>
<td>Category</td>
<td>Statement</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>“WHERE Statement” on page 1421</td>
<td>Selects observations from SAS data sets that meet a particular condition</td>
</tr>
<tr>
<td>Control</td>
<td>“CONTINUE Statement” on page 1125</td>
<td>Stops processing the current DO-loop iteration and resumes with the next iteration</td>
</tr>
<tr>
<td></td>
<td>“DO Statement” on page 1145</td>
<td>Designates a group of statements to be executed as a unit</td>
</tr>
<tr>
<td></td>
<td>“DO Statement, Iterative” on page 1146</td>
<td>Executes statements between DO and END repetitively based on the value of an index variable</td>
</tr>
<tr>
<td></td>
<td>“DO UNTIL Statement” on page 1150</td>
<td>Executes statements in a DO loop repetitively until a condition is true</td>
</tr>
<tr>
<td></td>
<td>“DO WHILE Statement” on page 1151</td>
<td>Executes statements repetitively while a condition is true</td>
</tr>
<tr>
<td></td>
<td>“END Statement” on page 1154</td>
<td>Ends a DO group or a SELECT group</td>
</tr>
<tr>
<td></td>
<td>“GO TO Statement” on page 1213</td>
<td>Moves execution immediately to the statement label that is specified</td>
</tr>
<tr>
<td></td>
<td>“IF-THEN/ELSE Statement” on page 1216</td>
<td>Executes a SAS statement for observations that meet specific conditions</td>
</tr>
<tr>
<td></td>
<td>“Labels, Statement” on page 1281</td>
<td>Identifies a statement that is referred to by another statement</td>
</tr>
<tr>
<td></td>
<td>“LEAVE Statement” on page 1282</td>
<td>Stops processing the current loop and resumes with the next statement in sequence</td>
</tr>
<tr>
<td></td>
<td>“LINK Statement” on page 1296</td>
<td>Jumps to a statement label</td>
</tr>
<tr>
<td></td>
<td>“RETURN Statement” on page 1387</td>
<td>Stops executing statements at the current point in the DATA step and returns to a predetermined point in the step</td>
</tr>
<tr>
<td></td>
<td>“SELECT Statement” on page 1396</td>
<td>Executes one of several statements or groups of statements</td>
</tr>
<tr>
<td>File-handling</td>
<td>“BY Statement” on page 1114</td>
<td>Controls the operation of a SET, MERGE, MODIFY, or UPDATE statement in the DATA step and sets up special grouping variables</td>
</tr>
<tr>
<td></td>
<td>“CARDS Statement” on page 1120</td>
<td>Indicates that data lines follow</td>
</tr>
<tr>
<td></td>
<td>“CARDS4 Statement” on page 1120</td>
<td>Indicates that data lines that contain semicolons follow</td>
</tr>
<tr>
<td></td>
<td>“DATA Statement” on page 1126</td>
<td>Begins a DATA step and provides names for any output SAS data sets</td>
</tr>
<tr>
<td></td>
<td>“DATALINES Statement” on page 1133</td>
<td>Indicates that data lines follow</td>
</tr>
<tr>
<td></td>
<td>“DATALINES4 Statement” on page 1134</td>
<td>Indicates that data lines that contain semicolons follow</td>
</tr>
<tr>
<td></td>
<td>“FILE Statement” on page 1157</td>
<td>Specifies the current output file for PUT statements</td>
</tr>
<tr>
<td>Category</td>
<td>Statement</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>“FILE, ODS Statement” on page 1171</td>
<td>Creates an ODS output object by binding the data component to the table definition (template). Optionally, lists the variables to include in the ODS output and specifies options that control the way that the variables are formatted.</td>
<td></td>
</tr>
<tr>
<td>“INFILE Statement” on page 1224</td>
<td>Identifies an external file to read with an INPUT statement</td>
<td></td>
</tr>
<tr>
<td>“INPUT Statement” on page 1248</td>
<td>Describes the arrangement of values in the input data record and assigns input values to the corresponding SAS variables</td>
<td></td>
</tr>
<tr>
<td>“INPUT Statement, Column” on page 1263</td>
<td>Reads input values from specified columns and assigns them to the corresponding SAS variables</td>
<td></td>
</tr>
<tr>
<td>“INPUT Statement, Formatted” on page 1265</td>
<td>Reads input values with specified informats and assigns them to the corresponding SAS variables</td>
<td></td>
</tr>
<tr>
<td>“INPUT Statement, List” on page 1269</td>
<td>Scans the input data record for input values and assigns them to the corresponding SAS variables</td>
<td></td>
</tr>
<tr>
<td>“INPUT Statement, Named” on page 1275</td>
<td>Reads data values that appear after a variable name that is followed by an equal sign and assigns them to corresponding SAS variables</td>
<td></td>
</tr>
<tr>
<td>“MERGE Statement” on page 1306</td>
<td>Joins observations from two or more SAS data sets into single observations</td>
<td></td>
</tr>
<tr>
<td>“MODIFY Statement” on page 1310</td>
<td>Replaces, deletes, and appends observations in an existing SAS data set in place; does not create an additional copy</td>
<td></td>
</tr>
<tr>
<td>“PUT Statement” on page 1344</td>
<td>Writes lines to the SAS log, to the SAS output window, or to an external location that is specified in the most recent FILE statement</td>
<td></td>
</tr>
<tr>
<td>“PUT Statement, Column” on page 1360</td>
<td>Writes variable values in the specified columns in the output line</td>
<td></td>
</tr>
<tr>
<td>“PUT Statement, Formatted” on page 1362</td>
<td>Writes variable values with the specified format in the output line</td>
<td></td>
</tr>
<tr>
<td>“PUT Statement, List” on page 1366</td>
<td>Writes variable values and the specified character strings in the output line</td>
<td></td>
</tr>
<tr>
<td>“PUT Statement, Named” on page 1370</td>
<td>Writes variable values after the variable name and an equal sign</td>
<td></td>
</tr>
<tr>
<td>“PUT, ODS Statement” on page 1373</td>
<td>Writes data values to a special buffer from which they can be written to the data component and formatted by ODS</td>
<td></td>
</tr>
<tr>
<td>“SET Statement” on page 1399</td>
<td>Reads an observation from one or more SAS data sets</td>
<td></td>
</tr>
<tr>
<td>“UPDATE Statement” on page 1416</td>
<td>Updates a master file by applying transactions</td>
<td></td>
</tr>
<tr>
<td>Information</td>
<td>“ARRAY Statement” on page 1102</td>
<td>Defines elements of an array</td>
</tr>
<tr>
<td></td>
<td>“Array Reference Statement” on page 1107</td>
<td>Describes the elements in an array to be processed</td>
</tr>
<tr>
<td>Category</td>
<td>Statement</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Category Statement</td>
<td>&quot;ATTRIB Statement&quot; on page 1111</td>
<td>Associates a format, informat, label, and/or length with one or more variables</td>
</tr>
<tr>
<td></td>
<td>&quot;DROP Statement&quot; on page 1152</td>
<td>Excludes variables from output SAS data sets</td>
</tr>
<tr>
<td></td>
<td>&quot;FORMAT Statement&quot; on page 1209</td>
<td>Associates formats with variables</td>
</tr>
<tr>
<td></td>
<td>&quot;INFORMAT Statement&quot; on page 1244</td>
<td>Associates informats with variables</td>
</tr>
<tr>
<td></td>
<td>&quot;KEEP Statement&quot; on page 1278</td>
<td>Includes variables in output SAS data sets</td>
</tr>
<tr>
<td></td>
<td>&quot;LABEL Statement&quot; on page 1280</td>
<td>Assigns descriptive labels to variables</td>
</tr>
<tr>
<td></td>
<td>&quot;LENGTH Statement&quot; on page 1283</td>
<td>Specifies the number of bytes for storing variables</td>
</tr>
<tr>
<td></td>
<td>&quot;MISSING Statement&quot; on page 1308</td>
<td>Assigns characters in your input data to represent special missing values for numeric data</td>
</tr>
<tr>
<td></td>
<td>&quot;RENAME Statement&quot; on page 1379</td>
<td>Specifies new names for variables in output SAS data sets</td>
</tr>
<tr>
<td></td>
<td>&quot;RETAIN Statement&quot; on page 1383</td>
<td>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</td>
</tr>
<tr>
<td>Window Display</td>
<td>&quot;DISPLAY Statement&quot; on page 1141</td>
<td>Displays a window that is created with the WINDOW statement</td>
</tr>
<tr>
<td></td>
<td>&quot;WINDOW Statement&quot; on page 1426</td>
<td>Creates customized windows for your applications</td>
</tr>
</tbody>
</table>

### 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 eight categories:

Table 7.3 Global Statements by Category

<table>
<thead>
<tr>
<th>Statements in this category …</th>
<th>let you …</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Access</td>
<td>associate reference names with SAS data libraries, SAS catalogs, external files and output devices, and access remote files.</td>
</tr>
<tr>
<td>Log Control</td>
<td>alter the appearance of the SAS log.</td>
</tr>
<tr>
<td>ODS: Output Control</td>
<td>choose objects to send to output destinations; edit the output format.</td>
</tr>
<tr>
<td>ODS: SAS Formatted</td>
<td>apply default styles to SAS specific entities such as a SAS data set, SAS output listing, or a SAS document.</td>
</tr>
<tr>
<td>ODS: Third-Party Formatted</td>
<td>apply styles to the output objects that are used by applications outside of SAS.</td>
</tr>
<tr>
<td>Operating Environment</td>
<td>access the operating environment directly.</td>
</tr>
<tr>
<td>Output Control</td>
<td>add titles and footnotes to your SAS output; deliver output in a variety of formats.</td>
</tr>
<tr>
<td>Program Control</td>
<td>govern the way SAS processes your SAS program.</td>
</tr>
</tbody>
</table>

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

Table 7.4 Categories and Descriptions of Global Statements

<table>
<thead>
<tr>
<th>Category</th>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Access</td>
<td>“CATNAME Statement” on page 1120</td>
<td>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</td>
</tr>
<tr>
<td></td>
<td>“FILENAME Statement” on page 1172</td>
<td>Associates a SAS fileref with an external file or an output device; disassociates a fileref and external file; lists attributes of external files</td>
</tr>
<tr>
<td></td>
<td>“FILENAME Statement, CATALOG Access Method” on page 1178</td>
<td>References a SAS catalog as an external file</td>
</tr>
<tr>
<td></td>
<td>“FILENAME, CLIPBOARD Access Method” on page 1181</td>
<td>Enables you to read text data from and write text data to the clipboard on the host machine.</td>
</tr>
<tr>
<td></td>
<td>“FILENAME Statement, EMAIL (SMTP) Access Method” on page 1183</td>
<td>Allows you to send electronic mail programmatically from SAS using the SMTP (Simple Mail Transfer Protocol) e-mail interface</td>
</tr>
<tr>
<td>Category</td>
<td>Statement</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>“FILENAME Statement, FTP Access Method” on page 1193</td>
<td>Enables you to access remote files by using the FTP protocol</td>
</tr>
<tr>
<td></td>
<td>“FILENAME Statement, SOCKET Access Method” on page 1201</td>
<td>Enables you to read from or write to a TCP/IP socket</td>
</tr>
<tr>
<td></td>
<td>“FILENAME Statement, URL Access Method” on page 1205</td>
<td>Enables you to access remote files by using the URL access method</td>
</tr>
<tr>
<td></td>
<td>“LIBNAME Statement” on page 1286</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>“LIBNAME Statement, SAS/ACCESS” on page 1294</td>
<td>Associates a SAS libref with a database management system (DBMS) database, schema, server, or group of tables or views</td>
</tr>
<tr>
<td></td>
<td>“LIBNAME Statement for SAS/CONNECT Remote Library Services” on page 1294</td>
<td>Associates a libref with a SAS data library that is located on the server for client access</td>
</tr>
<tr>
<td></td>
<td>“LIBNAME Statement for the Metadata Engine” on page 1294</td>
<td>Associates a SAS libref with the metadata that is in a SAS Metadata Repository on the SAS Metadata Server</td>
</tr>
<tr>
<td></td>
<td>“LIBNAME Statement for SAS/CONNECT TCP/IP Pipe” on page 1295</td>
<td>Associates a libref with a TCP/IP pipe (instead of a physical disk device) for processing input and output. The SASESOCK engine is required for SAS/CONNECT applications that implement MP CONNECT with piping.</td>
</tr>
<tr>
<td></td>
<td>“LIBNAME Statement for SAS/SHARE” on page 1295</td>
<td>In a client session, associates a libref with a SAS data library that is located on the server for client access. In a server session, pre-defines a server library that clients are permitted to access.</td>
</tr>
<tr>
<td></td>
<td>“LIBNAME Statement for Scalable Performance Data” on page 1295</td>
<td>Associates a SAS libref with a SAS data library for rapid processing of very large data sets by multiple CPUs</td>
</tr>
<tr>
<td></td>
<td>“LIBNAME Statement for the XML Engine” on page 1296</td>
<td>Associates a SAS libref with the physical location of an XML document</td>
</tr>
<tr>
<td>Log Control</td>
<td>“Comment Statement” on page 1123</td>
<td>Documents the purpose of the statement or program</td>
</tr>
<tr>
<td></td>
<td>“PAGE Statement” on page 1343</td>
<td>Skips to a new page in the SAS log</td>
</tr>
<tr>
<td></td>
<td>“SKIP Statement” on page 1406</td>
<td>Creates a blank line in the SAS log</td>
</tr>
<tr>
<td>ODS: Output Control</td>
<td>“LIBNAME Statement, SASEDOC” on page 1295</td>
<td>Associates a SAS libref with one or more ODS output objects that are stored in an ODS document</td>
</tr>
<tr>
<td>Category</td>
<td>Statement</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>“ODS <em>ALL</em> CLOSE Statement” on page 1332</td>
<td>Closes all open ODS output destinations, including the Listing destination</td>
</tr>
<tr>
<td></td>
<td>“ODS EXCLUDE Statement” on page 1333</td>
<td>Specifies output objects to exclude from ODS destinations</td>
</tr>
<tr>
<td></td>
<td>“ODS PATH Statement” on page 1335</td>
<td>Specifies which locations to search for definitions that were created by PROC TEMPLATE, as well as the order in which to search for them</td>
</tr>
<tr>
<td></td>
<td>“ODS PROCLABEL Statement” on page 1336</td>
<td>Enables you to change a procedure label</td>
</tr>
<tr>
<td></td>
<td>“ODS PROCTITLE Statement” on page 1337</td>
<td>Determines whether to suppress the writing of the title that identifies the procedure that produces the results</td>
</tr>
<tr>
<td></td>
<td>“ODS RESULTS Statement” on page 1337</td>
<td>Tracks ODS output in the Results window</td>
</tr>
<tr>
<td></td>
<td>“ODS SELECT Statement” on page 1338</td>
<td>Specifies output objects to send to the ODS destinations</td>
</tr>
<tr>
<td></td>
<td>“ODS SHOW Statement” on page 1338</td>
<td>Writes the specified selection or exclusion list to the SAS log</td>
</tr>
<tr>
<td></td>
<td>“ODS TRACE Statement” on page 1338</td>
<td>Writes to the SAS log a record of each output object that is created, or suppresses the writing of this record</td>
</tr>
<tr>
<td></td>
<td>“ODS USEGOPT Statement” on page 1339</td>
<td>Enables graphics option settings</td>
</tr>
<tr>
<td></td>
<td>“ODS VERIFY Statement” on page 1339</td>
<td>Writes or suppresses a warning that a style definition or a table definition that is used is not supplied by SAS</td>
</tr>
<tr>
<td></td>
<td>“ODS DECIMAL_ALIGN Statement” on page 1332</td>
<td>Aligns values by the decimal point in numeric columns when no justification is specified</td>
</tr>
<tr>
<td></td>
<td>“ODS DOCUMENT Statement” on page 1333</td>
<td>Creates an ODS document</td>
</tr>
<tr>
<td></td>
<td>“ODS LISTING Statement” on page 1334</td>
<td>Opens, manages, or closes the LISTING destination</td>
</tr>
<tr>
<td></td>
<td>“ODS OUTPUT Statement” on page 1335</td>
<td>Creates a SAS data set from an output object and manages the selection and exclusion lists for the Output destination</td>
</tr>
<tr>
<td></td>
<td>“ODS CHTML Statement” on page 1331</td>
<td>Produces a compact, minimal HTML that does not use style information</td>
</tr>
<tr>
<td></td>
<td>“ODS CSVALL Statement” on page 1332</td>
<td>Produces output that contains columns of data values that are separated by commas. ODS CSVALL produces tabular output with titles, notes, and bylines.</td>
</tr>
<tr>
<td></td>
<td>“ODS DOCBOOK Statement” on page 1332</td>
<td>Produces XML output that conforms to the DocBook DTD by OASIS</td>
</tr>
<tr>
<td></td>
<td>“ODS HTML Statement” on page 1333</td>
<td>Opens, manages, or closes the HTML destination, which produces HTML 4.0 embedded stylesheets</td>
</tr>
<tr>
<td></td>
<td>“ODS HTML3 Statement” on page 1334</td>
<td>Opens, manages, or closes the HTML3 destination, which produces HTML 3.2 formatted output</td>
</tr>
<tr>
<td>Category</td>
<td>Statement</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>“ODS HTMLCSS Statement” on page 1334</td>
<td>Produces HTML that is similar to ODS HTML with cascading style sheets</td>
<td></td>
</tr>
<tr>
<td>“ODS IMODE Statement” on page 1334</td>
<td>Produces HTML that is a column of output, separated by lines</td>
<td></td>
</tr>
<tr>
<td>“ODS MARKUP Statement” on page 1335</td>
<td>Opens, manages, or closes one or more specified destinations</td>
<td></td>
</tr>
<tr>
<td>“ODS PCL Statement” on page 1335</td>
<td>Provides portable support for PCL (HP LaserJet) files</td>
<td></td>
</tr>
<tr>
<td>“ODS PDF Statement” on page 1336</td>
<td>Opens, manages, or closes the PDF destination</td>
<td></td>
</tr>
<tr>
<td>“ODS PHTML Statement” on page 1336</td>
<td>Produces a basic HTML that uses twelve style elements and no class attributes</td>
<td></td>
</tr>
<tr>
<td>“ODS PRINTER Statement” on page 1336</td>
<td>Opens, manages, or closes the PRINTER destination</td>
<td></td>
</tr>
<tr>
<td>“ODS PS Statement” on page 1337</td>
<td>Opens, manages, or closes the PostScript destination</td>
<td></td>
</tr>
<tr>
<td>“ODS RTF Statement” on page 1338</td>
<td>Creates SAS output for Microsoft Word</td>
<td></td>
</tr>
<tr>
<td>“ODS WML Statement” on page 1339</td>
<td>Uses the Wireless Application Protocol (WAP) to produce a Wireless Markup Language (WML) DTD with a simple href list for a table of contents</td>
<td></td>
</tr>
<tr>
<td>Operating Environment</td>
<td>“X Statement” on page 1437</td>
<td>Issues an operating-environment command from within a SAS session</td>
</tr>
<tr>
<td>Output Control</td>
<td>“FOOTNOTE Statement” on page 1208</td>
<td>Writes up to 10 lines of text at the bottom of the procedure or DATA step output</td>
</tr>
<tr>
<td>Program Control</td>
<td>“DM Statement” on page 1143</td>
<td>Submits SAS Program Editor, Log, Procedure Output or text editor commands as SAS statements</td>
</tr>
<tr>
<td></td>
<td>“ENDSAS Statement” on page 1155</td>
<td>Terminates a SAS job or session after the current DATA or PROC step executes</td>
</tr>
<tr>
<td></td>
<td>“%INCLUDE Statement” on page 1218</td>
<td>Brings a SAS programming statement, data lines, or both, into a current SAS program</td>
</tr>
<tr>
<td></td>
<td>“%LIST Statement” on page 1300</td>
<td>Displays lines that are entered in the current session</td>
</tr>
<tr>
<td></td>
<td>“LOCK Statement” on page 1301</td>
<td>Acquires and releases an exclusive lock on an existing SAS file</td>
</tr>
<tr>
<td></td>
<td>“OPTIONS Statement” on page 1339</td>
<td>Changes the value of one or more SAS system options</td>
</tr>
<tr>
<td></td>
<td>“RUN Statement” on page 1389</td>
<td>Executes the previously entered SAS statements</td>
</tr>
</tbody>
</table>
ABORT Statement

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

**Valid:** in a DATA step  
**Category:** Action  
**Type:** Executable  
**See:** ABORT Statement in the documentation for your operating environment.

**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: The return code generated by the ABORT statement is ignored by SAS if the system option ERRORABEND is in effect. △

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.

```sas
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 Statement” on page 1407

ARRAY Statement

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: You can enclose the subscript in braces ({}), brackets ([ ]), or parentheses (( )).

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 elements.

**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 1, 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 1106

_**_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.

**Featured in:** Example 2 on page 1106, and Example 3 on page 1106

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

**Tip:** You can also use a shorthand notation for specifying a range of sequential integers. The increment is always +1.

**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 were named on 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.

**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_;
- array x{10} (2*1:5);

**Example 5: Creating a Range of Variable Names That Have Leading Zeroes**
The following example shows that you can create a range of variable names that have leading zeroes. Each variable name has a length of three characters, and the names sort correctly (A01, A02, ... A10). Without leading zeroes, the variable names would sort in the following order: A1, A10, A2, ... A9.

```sas
options pageno=1 nodate ps=64 ls=80;

data test (drop=i);
  array a(10) A01-A10;
  do i=1 to 10;
    a(i)=i;
  end;
run;
```
proc print noobs data=test;
runt;

Output 7.1  Array Names That Have Leading Zeroses

<table>
<thead>
<tr>
<th>The SAS System</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A01 A02 A03 A04 A05 A06 A07 A08 A09 A10</td>
<td></td>
</tr>
<tr>
<td>1  2  3  4  5  6  7  8  9  10</td>
<td></td>
</tr>
</tbody>
</table>

See Also

Statement:
“Array Reference Statement” on page 1107
“Array Processing” in SAS Language Reference: Concepts

Array Reference Statement

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.

Featured in: Example 1 on page 1109
Tip: You can enclose a subscript in braces ( {} ), 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.

Featured in: Example 4 on page 1109

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.

Featured in: Example 3 on page 1109

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:

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

```sas
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.

```sas
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:

```sas
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 Function" on page 498
- Statements: "ARRAY Statement" on page 1102
  "DO Statement, Iterative" on page 1146
- "Array Processing" in *SAS Language Reference: Concepts*
Assignment Statement

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 Statement” on page 1408
ATTRIB Statement

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

Valid: in a DATA step
Category: Information
Type: Declarative
See: ATTRIB Statement in the documentation for your operating environment.

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.

Restriction: You cannot change the length of a variable using LENGTH= from PROC DATASETS.
TRANSCODE=YES | NO
specifies whether character variables can be transcoded. Use TRANSCODE=NO to suppress transcoding. For more information about transcoding, see “Transcoding” in SAS National Language Support (NLS): User's Guide.

Default: YES

Restriction: Prior releases of SAS cannot access a SAS 9.1 data set that contains a variable with a TRANSCODE=NO attribute.

Restriction: Transcode suppression is not supported by the V6TAPE engine.

Interaction: You can use the VTRANSCODE and VTRANSCODEX functions to return a value that indicates whether transcoding is on or off for a character variable.

Interaction: If the TRANSCODE= attribute is set to NO for any character variable in a data set, then PROC CONTENTS prints a transcode column that contains the TRANSCODE= value for each variable in the data set. If all variables in the data set are set to the default TRANSCODE= value (YES), then no transcode column prints.

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.

How SAS Treats Transcoded Variables when You Use the SET and MERGE Statements  When you use the SET or MERGE statement to create a data set from several data sets, SAS makes the TRANSCODE= attribute of the variable in the output data set equal to the TRANSCODE= value of the variable in the first data set. See Example 2 on page 1113 and Example 3 on page 1113.

Note: The TRANSCODE= attribute is set when the variable is first seen on an input data set or in an ATTRIB TRANSCODE= statement. If a SET or MERGE statement comes before an ATTRIB TRANSCODE= statement and the TRANSCODE= attribute contradicts the SET statement, a warning will occur.
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

Example 1: Examples of ATTRIB Statements With Varying Numbers of Variables and Attributes

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';

Example 2: Using the SET Statement with Transcoded Variables

In this example, which uses the SET statement, the variable Z's TRANSCODE= attribute in data set A is NO because B is the first data set and Z's TRANSCODE= attribute in data set B is NO.

```plaintext
data b;
  length z $4;
  z = 'ice';
  attrib z transcode = no;
data c;
  length z $4;
  z = 'snow';
  attrib z transcode = yes;
data a;
set b;
set c;
/* Check transcode setting for variable Z */
rc1 = vtranscode(z);
pit rc1;
run;
```

Example 3: Using the MERGE Statement with Transcoded Variables

In this example, which uses the MERGE statement, the variable Z's TRANSCODE= attribute in data set A is YES because C is the first data set and Z's TRANSCODE= attribute in data set C is YES.
data b;
 length z $4;
 z = 'ice';
 attrib z transcode = no;
data c;
 length z $4;
 z = 'snow';
 attrib z transcode = yes;
data a;
 merge c b;
 /* Check transcode setting for variable Z */
 rcl = vtranscode(z);  
 put rcl=;
 run;

See Also

Statements:
“FORMAT Statement” on page 1209
“INFORMAT Statement” on page 1244
“LABEL Statement” on page 1280
“LENGTH Statement” on page 1283

Functions:
“VTRANSCODE Function” on page 910
“VTRANSCODEX Function” on page 911

BY Statement

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

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 data sets that are indexed because indexes are always stored in ascending order.

Featured in: Example 2 on page 1117

GROUPFORMAT
uses the formatted values, instead of the internal values, of the BY variables to determine where BY-groups begin and end, and therefore how FIRST:variable and LAST:variable are assigned. Although the GROUPFORMAT option can appear anywhere in the BY statement, the option applies to all variables in the BY statement.

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

Restriction: You can use the GROUPFORMAT option in a BY statement only in a DATA step.

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

Tip: Using the GROUPFORMAT option 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 without requiring that the data set be sorted or indexed.

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

See Also: By-Group Processing in the DATA Step in SAS Language Reference: Concepts

Featured in: Example 4 on page 1117

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 1117, Example 2 on page 1117, Example 3 on page 1117, and Example 4 on page 1117

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 the NOTSORTED option with the MERGE and UPDATE statements.

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

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

Featured in: Example 3 on page 1117

Details

How SAS Identifies the Beginning and End of a BY-Group SAS identifies the beginning and end of a BY group by creating two temporary variables for each BY variable: FIRST:variable and LAST:variable. The value of these variables are either 0 or 1. SAS sets the value of FIRST:variable to 1 when it reads the first observation in a BY group, and sets the value of LAST:variable to 1 when it reads the last observation in a BY
group. These temporary variables are available for DATA step programming but are not added to the output data set.

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

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 the Base SAS Procedures Guide for a discussion of how the BY statement affects processing for SAS procedures.

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

Processing BY-Groups SAS assigns the following values to FIRST.variable and LAST.variable:
- FIRST.variable has a value of 1 under the following conditions:
  - when the current observation is the first observation that is read from the data set.
  - when you do not use the GROUPFORMAT option and the internal value of the variable in the current observation differs from the internal value in the previous observation.

  If you use the GROUPFORMAT option, FIRST.variable has a value of 1 when the formatted value of the variable in the current observation differs from the formatted value in the previous observation.

  FIRST.variable has a value of 1 for any preceding variable in the BY statement.

In all other cases, FIRST.variable has a value of 0.

- LAST.variable has a value of 1 under the following conditions:
  - when the current observation is the last observation that is read from the data set.
  - when you use the GROUPFORMAT option and the internal value of the variable in the current observation differs from the internal value in the next observation.

  If you use the GROUPFORMAT option, LAST.variable has a value of 1 when the formatted value of the variable in the current observation differs from the formatted value in the previous observation.

  LAST.variable has a value of 1 for any preceding variable in the BY statement.
In all other cases, LAST:variable has a value of 0.

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 descending order of the values of JANSALES:
  
  by salesrep descending jansales;

- 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

The following example illustrates the use of the GROUPFORMAT option.

```plaintext
proc format;
  value range
    low -55 = 'Under 55'
    55-60 = '55 to 60'
    60-65 = '60 to 65'
    65-70 = '65 to 70'
    other = 'Over 70';
run;

proc sort data=class out=sorted_class;
  by height;
run;

data _null_
  format height range.;
  set sorted_class;
  by height groupformat;
  if first.height then
    put 'Shortest in ' height ' measures ' height:best12.;
run;
```
SAS writes the following output to the log:

<table>
<thead>
<tr>
<th>Category</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Under 55</td>
<td>51.3</td>
</tr>
<tr>
<td>55 to 60</td>
<td>56.3</td>
</tr>
<tr>
<td>60 to 65</td>
<td>62.5</td>
</tr>
<tr>
<td>65 to 70</td>
<td>65.3</td>
</tr>
<tr>
<td>Over 70</td>
<td>72</td>
</tr>
</tbody>
</table>

Example 5: Combining Multiple Observations and Grouping Them Based on One BY Value  
The following example shows how to use FIRST:variable and LAST:variable with BY group processing.

```sas
options pageno=1 nodate ls=80 ps=64;

data Inventory;
    length RecordID 8 Invoice $ 30 ItemLine $ 50;
    infile datalines;
    input RecordID Invoice ItemLine &;
    drop RecordID;
    datalines;
    A74  A5296  Highlighters
    A75  A5296  Lot # 7603
    A76  A5296  Yellow Blue Green
    A77  A5296  24 per box
    A78  A5297  Paper Clips
    A79  A5297  Lot # 7423
    A80  A5297  Small Medium Large
    A81  A5298  Gluestick
    A82  A5298  Lot # 4422
    A83  A5298  New item
    A84  A5299  Rubber bands
    A85  A5299  Lot # 7892
    A86  A5299  Wide width, Narrow width
    A87  A5299  1000 per box
;

data combined;
    array Line[4] $ 60 ;
    retain Line1-Line4;
    keep Invoice Line1-Line4;
    set Inventory;
    by Invoice;

    if first.Invoice then do;
        call missing(of Line1-Line4);
        records = 0;
    end;

    records + 1;
    Line[records]=ItemLine;

    if last.Invoice then output;
run;
```
proc print data=combined;
   title 'Office Supply Inventory';
run;

Output 7.2  Output from Combining Multiple Observations

<table>
<thead>
<tr>
<th>Obs</th>
<th>Line1</th>
<th>Line2</th>
<th>Line3</th>
<th>Line4</th>
<th>Invoice</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Highlighters</td>
<td>Lot # 7603</td>
<td>Yellow Blue Green</td>
<td>24 per box</td>
<td>A5296</td>
</tr>
<tr>
<td>2</td>
<td>Paper Clips</td>
<td>Lot # 7423</td>
<td>Small Medium Large</td>
<td></td>
<td>A5297</td>
</tr>
<tr>
<td>3</td>
<td>Gluestick</td>
<td>Lot # 4422</td>
<td>New item</td>
<td></td>
<td>A5298</td>
</tr>
<tr>
<td>4</td>
<td>Rubber bands</td>
<td>Lot # 7892</td>
<td>Wide width, Narrow width</td>
<td>1000 per box</td>
<td>A5299</td>
</tr>
</tbody>
</table>

See Also

Statements:
“MERGE Statement” on page 1306
“MODIFY Statement” on page 1310
“SET Statement” on page 1399
“UPDATE Statement” on page 1416

CALL Statement

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 243.

(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 Also
Chapter 4, “Functions and CALL Routines,” on page 243

---

**CARDS Statement**

Indicates that data lines follow

Valid: in a DATA step
Category: File-handling
Type: Declarative
Alias: DATALINES, LINES
See: “DATALINES Statement” on page 1133

---

**CARDS4 Statement**

Indicates that data lines that contain semicolons follow

Valid: in a DATA step
Category: File-handling
Type: Declarative
Alias: DATALINES4, LINES4
See: “DATALINES4 Statement” on page 1134

---

**CATNAME Statement**

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

Syntax

1. `CATNAME <libref.> catref < (libref.-1.catalog-1 <(ACCESS=READONLY)> <...libref.-n.catalog–n <(ACCESS=READONLY)>)> ;`

2. `CATNAME <libref.> catref CLEAR | _ALL_ CLEAR;`

3. `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 can be used for this catref.

`catalog` is the name of a catalog that is 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.

LISTwrites 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_ LISTwrites all catalog names that are included in any current catalog concatenation to the SAS log.

ACCESS=READONLYassigns 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 enables you to access entries in multiple catalogs by specifying a single catalog reference name (`libref.catref` or `catref`).
After you create a catalog concatenation, you can specify the catref in any context that accepts a simple (non-concatenated) 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**

You might need to access entries in several SAS catalogs. The most efficient way to access the information is to logically concatenate the catalogs. This enables access to the information without actually creating a new, separate, and possibly very large catalog.

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

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

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

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

The SAS log displays this message:
Output 7.3  Log Output from CATNAME Statement

NOTE: Catalog concatenation WORK.ALLCATS has been created.

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**  
After you create 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, you could work on private copies of your application entries by using PRIVATE_CATALOG. If you want to see how your entries function when they are combined with the public version of the application, you can use COMBINED_CATALOGS.

**See Also**

Statements:

“FILENAME Statement” on page 1172

“FILENAME Statement, CATALOG Access Method” on page 1178

“LIBNAME Statement” on page 1286 for a discussion of implicitly concatenating SAS catalogs.

---

**Comment Statement**

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 and are terminated with a semicolon.  
Restriction:  These comments must be written as separate statements.  
Restriction:  These comments cannot contain internal semicolons or unmatched quotation marks.  
Tip:  Inside a macro, a comment in this form is often disregarded:

*message;

When creating macros, therefore, use the PL/1–style comment:

/* message */

/*message*/  
specifies the text that explains or documents the statement or program.  
Range:  These comments can be any length.  
Tip:  These comments can contain semicolons and unmatched quotation marks.  
Tip:  You can write these comments within statements or anywhere a single blank can appear in your SAS code.  
Tip:  This is the only style of comment that can be used to actually comment out code.

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:
CONTINUE Statement

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 PT
Eye/Dental
Kelly 85111 PT
Smith 433 FT
HMO
;

See Also

Statements:
   “DO Statement, Iterative” on page 1146
   “LEAVE Statement” on page 1282

DATA Statement

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

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

Syntax

1. DATA <data-set-name-1 <(data-set-options-1)>>
   <. . .data-set-name-n <(data-set-options-n)>> </ DEBUG>;

2. DATA _NULL_;

3. 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)>;

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

5. DATA VIEW=view-name <(password-option)>;
   DESCRIPT;

6. 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 \textit{data-set-name} and that \textit{data-set-name} must match \textit{view-name}.

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

\textbf{Tip:} You can execute a DATA step without creating a SAS data set. See Example 5 on page 1132 for an example. For more information, see “When Not Creating a Data Set” on page 1129.

\textbf{See also:} For details about the types of SAS data set names and when to use each type, see “Names in the SAS Language” in \textit{SAS Language Reference: Concepts}.

\textbf{(data-set-options)}

specifies optional arguments that the DATA step applies when it writes observations to the output data set.

\textbf{See also:} “Definition of Data Set Options” on page 6 for more information.

\textbf{Featured in:} Example 1 on page 1131

/ DEBUG

enables you to debug your program interactively by helping to identify logic errors, and sometimes data errors.

\_NULL\

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

\textbf{VIEW=}view-name

names a view that the DATA step uses to store the input DATA step view.

\textbf{Restriction:} \textit{view-name} must match one of the data set names.

\textbf{Restriction:} SAS creates only one view in a DATA step.

\textbf{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.

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

\textbf{Featured in:} Example 2 on page 1131 and Example 3 on page 1131

password-option

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

\textbf{ALTER=}alter-password

assigns an \textit{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.

\textbf{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

specifies 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 1131

---

**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.

**Creating 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.

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

**When 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.

```sas
data _NULL_;
  set sample;
  put Name ID;
run;
```
Creating 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.

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

For more information about DATA step views, see “SAS Data Views” in SAS Language Reference: Concepts.

Creating 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.

```sas
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 “Stored Compiled DATA Step Programs” in SAS Language Reference: Concepts.

Describing 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.

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

For information about the DESCRIBE statement, see the “DESCRIBE Statement” on page 1141.

Executing 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.

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

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

For information about the DESCRIBE statement, see the “DESCRIBE Statement” on page 1141. For information about the EXECUTE statement, see the “EXECUTE Statement” on page 1156.
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:

```sas
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:

```sas
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:

```sas
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:

```sas
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.

```sas
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:

```sas
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.

```sas
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.

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

See Also

Statements:

“DESCRIBE Statement” on page 1141
“EXECUTE Statement” on page 1156
“Definition of Data Set Options” on page 6
**DATALINES Statement**

- **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 Statement” on page 1134 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 System Option” on page 1489 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 1134.

**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:

```sas
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.

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

**See Also**

Statements:

- “DATALINES4 Statement” on page 1134
- “INFILE Statement” on page 1224

System Option:

- “CARDIMAGE System Option” on page 1489

---

**DATALINES4 Statement**

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 Statement” on page 1133

---

**DECLARE Statement**

Declares a DATA step component object; creates an instance of and initializes data for a DATA step component object

**Valid:** in a DATA step

**Category:** Action

**Type:** Executable

**Alias:** DCL
Syntax

DECLARE object variable<(argument_tag-1: value-1, ...argument_tag-n: value-n)>;

Arguments

object
specifies the component object. It can be one of the following:

hash
indicates a hash object. The hash object provides a mechanism for quick data storage and retrieval. The hash object stores and retrieves data based on look-up keys.

See Also: “Using the Hash Object” in SAS Language Reference: Concepts

hiter
indicates a hash iterator object. The hash iterator object enables you to retrieve the hash object's data in forward or reverse key order.

See Also: “Using the Hash Iterator Object” in SAS Language Reference: Concepts

variable
specifies the variable name for the component object.

argument_tag
specifies the information that is used to create an instance of the component object. Valid values for the argument_tag depend on the component object.

Valid hash object argument tags are

hashexp: n
The hash object’s internal table size, where the size of the hash table is $2^n$.

The value of hashexp is used as a power-of-two exponent to create the hash table size. For example, a value of 4 for hashexp equates to a hash table size of $2^4$, or 16. The maximum value for hashexp is 16, which equates to a hash table size of $2^{16}$ or 65536.

The hash table size is not equal to the number of items that can be stored. Imagine the hash table as an array of 'buckets.' A hash table size of 16 would have 16 'buckets.' Each bucket can hold an infinite number of items. The efficiency of the hash table lies in the ability of the hashing function to map items to and retrieve items from the buckets.

You should specify the hash table size relative to the amount of data in the hash object in order to maximize the efficiency of the hash object lookup routines. Try different hashexp values until you get the best result. For example, if the hash object contains one million items, a hash table size of 16 (hashexp = 4) would work, but not very efficiently. A hash table size of 512 or 1024 (hashexp = 9 or 10) would result in the best performance.

Default: 8, which equates to a hash table size of $2^4$ or 256

dataset: 'dataset_name'
Names a SAS data set to load into the hash object.

The name of the SAS data set can be a literal or character variable. The data set name must be enclosed in single or double quotation marks. Macro variables must be enclosed in double quotation marks.

Note: If the data set contains duplicate keys, then the first instance will be in the hash object, and following instances will be ignored.
ordered: `option`  
Specifies whether or how the data is returned in key-value order if you use the hash object with a hash iterator object or if you use the hash object OUTPUT method.

`option` can be one of the following values:

- `'ascending'` | `a` Data is returned in ascending key-value order. Specifying `ascending` is the same as specifying `'yes'`.
- `'descending'` | `d` Data is returned in descending key-value order.
- `'YES'` | `Y` Data is returned in ascending key-value order. Specifying `yes` is the same as specifying `ascending`.
- `'NO'` | `N` Data is returned in some undefined order.

**Default:** NO

The argument can also be enclosed in double quotation marks.

The valid hash iterator object constructor is the hash object name.

**See Also:** “Initializing Hash Object Data Using a Constructor” and “Declaring and Instantiating a Hash Iterator Object” in *SAS Language Reference: Concepts*.

**Details**

**The Basics**

To use a DATA step component object in your SAS program, you must declare and create (instantiate) the object. The DATA step component interface provides a mechanism for accessing predefined component objects from within the DATA step.

For more information about the predefined DATA step component objects, see “Using DATA Step Component Objects” in *SAS Language Reference: Concepts*.

**Declaring a DATA Step Component Object**

You use the DECLARE statement to declare the DATA step component object.

```sas
declare hash h;
```

The DECLARE statement tells SAS that the variable H is a hash object.

After you declare the new component object, use the `_NEW_` statement to instantiate the object. For example, in the following line of code, `_NEW_` statement creates the hash object and assigns it to the variable H:

```sas
h = _new_ hash();
```

**Using the DECLARE Statement to Instantiate a DATA Step Component Object**

As an alternative to the two-step process of using the DECLARE and the `_NEW_` statements to declare and instantiate a component object, you can use the DECLARE statement to declare and instantiate the component object in one step. For example, in the following line of code, the DECLARE statement declares and instantiates a hash object and assigns it to the variable H:

```sas
declare hash h();
```

This is equivalent to using the following code:

```sas
declare hash h;

h = _new_ hash();
```

A constructor is a method that you can use to instantiate a component object and initialize the component object data. For example, in the following line of code, the DECLARE statement declares and instantiates a hash object and assigns it to the
variable H. In addition, the hash table size is initialized to a value of 16 ($2^4$) using the argument tag, hashexp.

```declare hash h(hashexp: 4);```

**Comparisons**

You can use the DECLARE statement and the _NEW_ statement, or the DECLARE statement alone to declare and instantiate an instance of a DATA step component object.

**Examples**

**Example 1: Declaring and Instantiating a Hash Object by Using the DECLARE and _NEW_ Statements**

This example uses the DECLARE statement to declare a hash object. The _NEW_ statement is used to instantiate the hash object.

```data _null_;  
length k $15;  
length d $15;  
if _N_ = 1 then do;  
    /* Declare and instantiate hash object "myhash" */  
    declare hash myhash;  
    myhash = _new_ hash( );  
    /* Define key and data variables */  
    rc = myhash.defineKey('k');  
    rc = myhash.defineData('d');  
    rc = myhash.defineDone( );  
    /* avoid uninitialized variable notes */  
    call missing(k, d);  
end;  
/* Create constant key and data values */  
rc = myhash.add(key: 'Labrador', data: 'Retriever');  
rc = myhash.add(key: 'Airedale', data: 'Terrier');  
rc = myhash.add(key: 'Standard', data: 'Poodle');  
/* Find data associated with key and write data to log*/  
rc = myhash.find(key: 'Airedale');  
if (rc = 0) then  
    put d=;  
else  
    put 'Key Airedale not found';  
run;```

**Example 2: Declaring and Instantiating a Hash Object by Using the DECLARE Statement**

This example uses the DECLARE statement to declare and instantiate a hash object in one step.

```data _null_;  
length k $15;  
length d $15;  
if _N_ = 1 then do;  
    /* Declare and instantiate hash object "myhash" */  
    declare hash myhash( );  
    rc = myhash.defineKey('k');  
    rc = myhash.defineData('d');  
    rc = myhash.defineDone( );  
    /* avoid uninitialized variable notes */```
Example 3: Instantiating and Initializing Data Values for a Hash Object

This example uses the DECLARE statement to declare and instantiate a hash object. The hash table size is set to 16 ($2^4$).

```sas
data _null_;  
    length k $15;  
    length d $15;  
    if _N_ = 1 then do;  
        /* Declare and instantiate hash object "myhash". Set hash table size to 16.*/  
        declare hash myhash (hashexp: 4);  
        rc = myhash.defineKey('k');  
        rc = myhash.defineData('d');  
        rc = myhash.defineDone();  
        /* avoid uninitialized variable notes */  
        call missing(k, d);  
    end;  
    /* Create constant key and data values */  
    rc = myhash.add(key: 'Labrador', data: 'Retriever');  
    rc = myhash.add(key: 'Airedale', data: 'Terrier');  
    rc = myhash.add(key: 'Standard', data: 'Poodle');  
    /* Find data associated with key and write data to log*/  
    rc = myhash.find(key: 'Airedale');  
    if (rc = 0) then  
        put d=;  
    else  
        put 'Key Airedale not found';  
    run;
```

See Also

Statements:

“_NEW_ Statement” on page 1327
DELETE Statement

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:
DESCRIBE Statement

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 Statement” on page 1126.

See Also

Statements:
“DATA Statement” on page 1126
“EXECUTE Statement” on page 1156

DISPLAY Statement

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” on page 1426 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 Statement” on page 1426

DM Statement

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 “Details” on page 1144.
**Default:** If you omit the window name, SAS uses the Program Editor window as the default.

*command*

- "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.

```plaintext
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.

```sas
dm 'af c=your-program' continue;
proc print data=temp;
run;
```

---

DO Statement

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.

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

**See Also**

Statements:

“DO Statement, Iterative” on page 1146
“DO UNTIL Statement” on page 1150
“DO WHILE Statement” on page 1151

---

**DO Statement, 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 iterative 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 1148

**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 1148

**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 1148

**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 1148

**See Also:** “DO WHILE Statement” on page 1151 and “DO UNTIL Statement” on page 1150 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, 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 executes ten times. The first example demonstrates the preferred approach.

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

The next example uses the TO and BY arguments.
do i=1 to n by m;
  ...more SAS statements...
  if i=10 then leave;
end;
if i=10 then put 'EXITED LOOP';

Example 3: Stopping the 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 Statement” on page 1102
“Array Reference Statement” on page 1107
“DO Statement” on page 1145
“DO UNTIL Statement” on page 1150
“DO WHILE Statement” on page 1151
“GO TO Statement” on page 1213

---

DO UNTIL Statement

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 Statement” on page 1145
“DO Statement, Iterative” on page 1146
“DO WHILE Statement” on page 1151

DO WHILE Statement

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).

\[
\begin{align*}
n &= 0; \\
do & \text{ while}(n<5); \\
& \quad \text{put } n=; \\
& \quad n+1; \\
& \text{end;}
\end{align*}
\]

See Also

Statements:
- “DO Statement” on page 1145
- “DO Statement, Iterative” on page 1146
- “DO UNTIL Statement” on page 1150

DROP Statement

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 INVENTORY:

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

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:

\[
\begin{align*}
\text{\# do; } & \quad \ldots \ldots \text{more SAS statements. } \\
& \quad \text{end;} \\
\text{\# select(expression);} & \quad \text{when(expression) SAS statement;} \\
& \quad \text{otherwise SAS statement;} \\
& \quad \text{end;}
\end{align*}
\]

See Also

Statements:

“DO Statement” on page 1145

“SELECT Statement” on page 1396
ENDSAS Statement

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

“SYSSTARTID Automatic Macro Variable” in SAS Macro Language: Reference

ERROR Statement

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.

```sas
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.

```sas
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 Statement” on page 1344

EXECUTE Statement
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 Statement” on page 1126.

See Also
Statements:
“DATA Statement” on page 1126
“DESCRIBE Statement” on page 1141

FILE Statement
Specifies the current output file for PUT statements
Valid: in a DATA step
Category: File-handling
Type: Executable
See: FILE Statement in the documentation for your operating environment.

Syntax
FILE file-specification <options> <operating-environment-options>;
Arguments

file-specification

defines 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 Statement” on page 1172

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 Statement” on page 1172

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 file as the output that is produced by SAS procedures.

Interaction: When you write to a 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 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: Dependent on your operating environment.

Operating Environment Information: 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='string-in-quotations' | character-variable
specifies an alternate delimiter (other than blank) to be used for LIST output. This option is ignored for other types of output (formatted, column, named).

Alias: DLM=

Default: blank space

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 delimiter sensitive data (DSD) option.

Tip: DELIMITER= can be used with the colon (:) modifier (modified LIST output).

See Also: DSD (delimiter sensitive data) option on page 1159

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 exceeds the current line length is 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 might 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 was 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 1161 and STOPOVER on page 1164

DSD (delimiter sensitive data)
specifies that data values that contain embedded delimiters, such as tabs or commas, be enclosed in quotation marks. The DSD option enables you to write data values that contain embedded delimiters to LIST output. This option is ignored for other types of output (formatted, column, named). Any double
quotation marks that are included in the data value are promoted (repeated). When a variable value contains the delimiter and DSD is used in the FILE statement, the variable value will be enclosed in double quotation marks when the output is generated. For example, the following code

```sas
DATA _NULL_
  FILE log dsd;
  x=""lions, tigers, and bears"";
  put x ' "Oh, my!"';
run;
```

will result in the following output:

""lions, tigers, and bears"", "Oh, my!"

If a quoted (text) string contains the delimiter and DSD is used in the FILE statement, then the quoted string will not be enclosed in double quotation marks when used in a PUT statement. For example, the following code

```sas
DATA _NULL_
  FILE log dsd;
  PUT 'lions, tigers, and bears';
run;
```

will result in the following output:

lions, tigers, and bears

**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 values that do not contain the delimiter that you specify are not enclosed in quotation marks. However, you can use the tilde (~) modifier to force any data value, including missing values, to be enclosed in quotation marks, even if it contains no embedded delimiter.

**See Also:** DELIMITER= on page 1159

**ENCODING= 'encoding-value'**
specifies the encoding to use when writing to the output file. The value for ENCODING= indicates that the output file has a different encoding from the current session encoding.

When you write data to the output file, SAS transcodes the data from the session encoding to the specified encoding.


**Default:** SAS uses the current session encoding.

**Featured in:** Example 8 on page 1171

**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 1161

**Featured in:** Example 4 on page 1169
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 1160
Featured in: Example 5 on page 1169

FLOWOVER
causes data that exceeds 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 1159 and STOPOVER on page 1164

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 1168

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 file that contains carriage control characters or to the SAS log.
- the LRECL= option in the FILE statement when you write to a 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 1162, DROPOVER on page 1159, FLOWOVER on page 1161, and STOPOVER on page 1164

Featured in: Example 6 on page 1170

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 1168

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

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

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 1162, PAD on page 1164, and PAGESIZE= on page 1164
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 1164

**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 printed.  

**Restriction:** If the current output file is a file that is to be printed, *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 1168

**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 holds the results of the DATA step 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 output appropriately. For information about the ODS-suboptions, see the “FILE Statement for ODS”. For general information about the Output Delivery System, see SAS Output Delivery System: User’s Guide.

**Default:** If you omit the ODS suboptions, the DATA step uses a default table definition *(base.datastep.table)* that is stored in the SASHELP.TMPLMST template store. 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 _FILE_=, FILEVAR=, HEADER=, and PAD 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 have an effect only 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 1162

**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 1162

**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= System Option” on page 1581

**PRINT | NOPRINT**

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

*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.

**Restriction:** When you write to a 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.

**RECFM=record-format**

specifies the record format of the output file.

**Range:** Values are dependent on the operating environment.

*Operating Environment Information:* 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 1161 and DROPOVER on page 1159

**TITLES | NOTITLES**
controls the printing of the current title lines on the pages of 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:** “Updating the _FILE_ Variable” on page 1166

### Operating Environment Options

**Operating Environment Information:** For descriptions of operating-environment-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 1164.
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 operating-environment-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 the “FILE Statement for ODS” in SAS Output Delivery System: User’s Guide.

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 INFILE and FILE 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 INFILE and FILE 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 variables, _FILE_ is not written to the data set.

When you specify the _FILE_= option in a FILE statement, 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 cannot 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_.

Updating the _FILE_ Variable Like other SAS variables, you can update the _FILE_ variable. The following two methods are available:

- Use _FILE_ in an assignment statement.
- Use a PUT statement.

You can update the _FILE_ variable by using an assignment statement that has the following form.

_FILE_ = <'string-in-quotation-marks' | character-expression>

The assignment statement updates the contents of the current output buffer and sets the buffer length to the length of 'string-in-quotation-marks' or character-expression. However, this does not affect the current column pointer of the PUT statement.
next PUT statement for this FILE statement begins to update the buffer at column 1 or at the last known location when you use the trailing @ in the PUT statement.

In the following example, the assignment statement updates the contents of the current output buffer. The column pointer of the PUT statement is not affected:

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

SAS creates the following output:

**This is PUT**

In this example,

```sas
file print;
_file_ = 'This is from FILE, sir.';
put @14 'both';
```

SAS creates the following output:

**This is from both, sir.**

You can also update the _FILE_ variable by using a PUT statement. The PUT statement updates 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 want 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 example, when N=1

```sas
file ABC;
put 'Something' @;
Y = _file_||' is here';
file ABC;
put 'Nothing' ;
Y = _file_||' is here';
```

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

Any modification of _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 Beginning a New Page  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:
  ```sas
  data _null_;  
  set sprint;  
  by dept;  
  
  Route output to the SAS output window. 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:
  ```sas
  file print header=newpage;  
  ```

- Start a new page for each department:
  ```sas
  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:
  ```sas
  return;  
  
  newpage:  
  put @20 'Sales for 1989' /  
  @20 dept=;  
  return;  
  ```

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:
  ```sas
  data _null_;  
  set info;  
  ```

- Route output to the standard SAS output window. 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:
  ```sas
  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:
  ```sas
  if remain<7 then put _page_;  
  ```

Example 3: Arranging the Contents of an Entire Page  This example shows how to use 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 SAS output window. Use DATA _NULL_ to write a report rather than create a data set. PRINT is the fileref. SAS uses carriage control characters to write the output with the characteristics of a print file. N=PAGESIZE makes the entire page available to the output pointer:

```sas
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:

```sas
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:

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

After composing two 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:

```sas
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:

```sas
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:

```sas
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:

```sas
data _null_;  
  length name $ 200;
```
Read an in-stream data line and assign a value to the NAME variable:

```sas
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:

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

Append a log record to currently open output file:

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

Supply the names of the external files:

```sas
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:

```sas
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:

```sas
/* 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:

```sas
/* 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;

Example 8: Specifying an Encoding When Writing to an Output File

This example creates an external file from a SAS data set. The current session encoding is WLatin1, but the external file's encoding needs to be UTF-8. By default, SAS writes the external file using the current session encoding.

To tell SAS what encoding to use when writing data to the external file, specify the ENCODING= option. When you tell SAS that the external file is to be in UTF-8 encoding, SAS then transcodes the data from WLatin1 to the specified UTF-8 encoding when writing to the external file.

libname myfiles 'SAS-data-library';

filename outfile 'external-file';

data _null_
set myfiles.cars;
file outfile encoding="utf-8";
put Make Model Year;
run;

See Also

Statements:
“FILENAME Statement” on page 1172
“INFILE Statement” on page 1224
“LABEL Statement” on page 1280
“PUT Statement” on page 1344
“RETURN Statement” on page 1387
“TITLE Statement” on page 1409

FILE, ODS Statement

Creates an ODS output object by binding the data component to the table definition (template). Optionally, lists the variables to include in the ODS output and specifies options that control the way that the variables are formatted.

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.
FILENAME Statement

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
See: FILENAME Statement in the documentation for your operating environment

Syntax

FILENAME fileref <device-type> 'external-file' <ENCODING='encoding-value'> <options><operating-environment-options>;
FILENAME fileref <device-type><options> <operating-environment-options>;
FILENAME fileref CLEAR | _ALL_ CLEAR;
FILENAME fileref LIST | _ALL_ LIST;

Arguments

fileref
is any SAS name that you use when you assign a new fileref. When you disassociate a currently assigned fileref or when you list file attributes with the FILENAME statement, specify a fileref that was previously assigned with a FILENAME statement or an operating environment-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 by using 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 about 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.

ENCODING= 'encoding-value'
specifies the encoding to use when SAS is reading from or writing to an external file. The value for ENCODING= indicates that the external file has a different encoding from the current session encoding.

See Also: The FILE, ODS statement in SAS Output Delivery System: User’s Guide
When you read data from an external file, SAS transcodes the data from the specified encoding to the session encoding. When you write data to an external file, SAS transcodes the data from the session encoding to the specified encoding. For valid encoding values, see “Encoding Values in SAS Language Elements” in SAS National Language Support (NLS): User’s Guide.

**Default:** SAS assumes that an external file is in the same encoding as the session encoding.

**Featured in:** Example 5 on page 1177 and Example 6 on page 1177

**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.

- **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 receive 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.

- **UPRINTER** specifies a Universal Printing printer definition name.

**Operating Environment Information:** Additional specifications might 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 might 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.

Options

RECFM=record-format
specifies the record format of the external file.

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

Operating Environment Options

Operating environment 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

Operating Environment Information

Using the FILENAME statement requires operating environment-specific information. See the SAS documentation for your operating environment before using this statement. Note also that 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. See Example 3 on page 1176.

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 about specifying external files, see the SAS documentation for your operating environment.

fileref
(a file reference name) is a shorthand reference to an external file. After you associate 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.

Associating 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’ <operating-environment-options>;}
To associate a fileref with a file other than a disk file, you might need to specify a device type, depending on your operating environment, as shown in this form:

FILENAME fileref <device-type> <operating-environment-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.

To specify a character-set encoding, use the following form:

FILENAME fileref <device-type> <operating-environment-options>;

**Associating a Fileref with a Terminal, Printer, Universal Printer, or Plotter**

To associate a fileref with an output device, use this form:

FILENAME fileref device-type <operating-environment-options>;

**Disassociating a Fileref from an External File**

To disassociate a fileref from a file, use a FILENAME statement, specifying the fileref and the CLEAR option.

**Writing 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 the 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 Filename**

You can specify an external file either by associating a fileref with the file and then specifying the fileref or by specifying the physical filename in quotation marks:

```sas
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 quotation marks */
   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.

```sas
filename green 'your-input-file';
libname save 'SAS-data-library';
data save.vegetable;
  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 that are stored there.

```
Operating Environment Information: Some operating environments allow you to read from but not write to members of aggregate storage locations. For details, see the SAS documentation for your operating environment.

In this example, each DATA step reads from an external file (REGION1 and 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 an operating environment-dependent option. The FILE statement directs PUT statement output to that printer.

```
filename out printer operating-environment-option;
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
```
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:

```sas
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
;
```

**Example 5: Specifying an Encoding When Reading an External File** This example creates a SAS data set from an external file. The external file is in UTF-8 character-set encoding, and the current SAS session is in the Wlatin1 encoding. By default, SAS assumes that an external file is in the same encoding as the session encoding, which causes the character data to be written to the new SAS data set incorrectly.

To tell SAS what encoding to use when reading the external file, specify the ENCODING= option. When you tell SAS that the external file is in UTF-8, SAS then transcodes the external file from UTF-8 to the current session encoding when writing to the new SAS data set. Therefore, the data is written to the new data set correctly in Wlatin1.

```sas
libname myfiles ‘SAS-data-library’;
filename extfile ‘external-file’ encoding=”utf-8”;
data myfiles.unicode;
  infile extfile;
  input Make $ Model $ Year;
run;
```

**Example 6: Specifying an Encoding When Writing to an External File** This example creates an external file from a SAS data set. The current session encoding is Wlatin1, but the external file's encoding needs to be UTF-8. By default, SAS writes the external file using the current session encoding.

To tell SAS what encoding to use when writing data to the external file, specify the ENCODING= option. When you tell SAS that the external file is to be in UTF-8 encoding, SAS then transcodes the data from Wlatin1 to the specified UTF-8 encoding when writing to the external file.

```sas
libname myfiles ‘SAS-data-library’;
filename outfile ‘external-file’ encoding=”utf-8”;
data _null_;   set myfiles.cars;
```
FILE outfile;
   put Make Model Year;
run;

See Also

Statements:
   "FILE Statement" on page 1157
   "%INCLUDE Statement" on page 1218
   "INFILE Statement" on page 1224
   "FILENAME Statement, CATALOG Access Method" on page 1178
   "FILENAME Statement, EMAIL (SMTP) Access Method" on page 1183
   "FILENAME Statement, FTP Access Method" on page 1193
   "FILENAME Statement, SOCKET Access Method" on page 1201
   "FILENAME Statement, URL Access Method" on page 1205
   "LIBNAME Statement" on page 1286

SAS Windowing Interface Commands:
FILE and INCLUDE

FILENAME Statement, 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.

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 1180.

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.
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;
runc;

  /* 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;';
runc;

%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:
FILENAME, CLIPBOARD Access Method

Enables you to read text data from and write text data to the clipboard on the host machine

Valid: anywhere
Category: Data Access

Syntax
FILENAME fileref CLIPBRD <BUFFER=paste-buffer-name>;

Arguments

fileref
is a valid fileref.

CLIPBRD
specifies the access method that enables you to read data from or write data to the clipboard on the host machine.

BUFFER=paste-buffer-name
creates and names the paste buffer. You can create any number of paste buffers by naming them with the BUFFER= argument in the STORE command.

Details
The FILENAME statement, CLIPBOARD Access Method enables you to share data within SAS and between SAS and applications other than SAS.

Comparisons
The STORE command copies marked text in the current window and stores the copy in a paste buffer.
You can also copy data to the clipboard by using the Explorer pop-up menu item **Copy Contents to Clipboard**.

**Examples**

**Example 1: Using ODS to Write a Data Set as HTML to the Clipboard**  This example uses the Sashelp.Air data set as the input file. The ODS is used to write the data set in HTML format to the clipboard.

```plaintext
filename _temp_ clipbrd;
  ods noresults;
  ods listing close;
  ods html file=_temp_ rs=none style=minimal;
  proc print data=Sashelp.'Air'N noobs;
run;
ods html close;
ods results;
ods listing;
filename _temp_;
```

**Example 2: Using the DATA Step to Write a Data Set As Comma-separated Values to the Clipboard**  This example uses the Sashelp.Air data set as the input file. The data is written in the DATA step as comma-separated values to the clipboard.

```plaintext
filename _temp1_ temp;
filename _temp2_ clipbrd;
proc contents data=Sashelp."Air"N out=info noprint;
proc sort data=info;
  by npos;
run;

data _null_;  
  set info end=eof;
  file _temp1_ dsd;
  put name @@;
  if _n_=1 then do;
    call execute("data _null_; set Sashelp."`Air`N; file _temp1_ dsd mod; put");
  end;
  call execute(trim(name));
  if eof then call execute('; run;');
run;

data _null_;  
  infile _temp1_;  
  file _temp2_;  
  input;  
  put _infile_;  
run;

filename _temp1_ clear;
filename _temp2_ clear;
```

**Example 3: Using the DATA Step to Write Text to the Clipboard**  This example writes three lines to the clipboard.
Example 4: Using the DATA Step to Retrieve Text from the Clipboard
This example writes three lines to the clipboard and then retrieves them.

```sas
filename clippy clipbrd;

data _null_; file clippy;
   put 'Line 1';
   put 'Line 2';
   put 'Line 3';
run;

data _null_; infile clippy;
   input;
   put _infile_;
run;
```

See Also

Command:
The STORE command in the Base SAS Help and Documentation.

---

FILENAME Statement, EMAIL (SMTP) Access Method

Allows you to send electronic mail programmatically from SAS using the SMTP (Simple Mail Transfer Protocol) e-mail interface

Valid: Anywhere
Category: Data Access

Syntax

```sas
FILENAME fileref EMAIL <'address' ><email-options>;
```
Arguments

fileref
is a valid file reference. The fileref is a name that is temporarily assigned to an external file or to a device type. Note that the fileref cannot exceed eight characters.

EMAIL
specifies the EMAIL device type, which provides the access method that enables you to send electronic mail programmatically from SAS. In order to use SAS to send a message to an SMTP server, you must enable SMTP e-mail. For more information, see “The SMTP E-Mail Interface” in SAS Language Reference: Concepts.

’address’
is the e-mail address to which you want to send the message. You must enclose the address in quotation marks. Specifying an address as a FILENAME statement argument is optional if you specify the TO= e-mail option or the PUT statement !EM_TO! directive, which will override an address specification.

E-mail Options
You can use any of the following email options in the FILENAME statement to specify attributes for the electronic message.

Note: You can also specify these options in the FILE statement. E-mail options that you specify in the FILE statement override any corresponding e-mail options that you specified in the FILENAME statement.

CONTENT_TYPE='content/type'
specifies the content type for the message body. If you do not specify a content type, SAS tries to determine the correct content type. You must enclose the value in quotation marks.

Aliases: CT= and TYPE=

Default: text/plain

ENCODING='encoding-value'
specifies the text encoding to use for the message body. For valid encoding values, see “Encoding Values in SAS Language Elements” in SAS National Language Support (NLS): User’s Guide.

TO='to-address'
specifies the primary recipient(s) of the e-mail message. You must enclose the address in quotation marks. To specify more than one address, enclose the group of addresses in parentheses, enclose each address in quotation marks, and separate each address with a space. To specify a real name as well as an address, enclose the address in angle brackets (< >). Here are examples:

to='joe@site.com'

to=('joe@site.com" jane@home.net")

to="Joe Smith <joe@site.com>"

Tip: Specifying TO= overrides the ’address’ argument.
CC='cc-address'
specifies the recipient(s) to receive a copy of the e-mail message. You must enclose an address in quotation marks. To specify more than one address, enclose the group of addresses in parentheses, enclose each address in quotation marks, and separate each address with a space. To specify a real name as well as an address, enclose the address in angle brackets (<>). Here are examples:

    cc='joe@site.com'

    cc=('joe@site.com' "jane@home.net")

    cc="Joe Smith <joe@site.com>"

BCC='bcc-address'
specifies the recipient(s) that you want to receive a blind copy of the electronic mail. Individuals that are listed in the bcc field will receive a copy of the e-mail. The BCC field does not appear in the e-mail header, so that these e-mail addresses cannot be viewed by other recipients.

    If a BCC address contains more than one word, then enclose it in quotation marks. To specify more than one address, you must enclose the group of addresses in parentheses, enclose each address in quotation marks, and separate each address with a space. To specify a real name as well as an address, enclose the address in angle brackets (<>). Here are examples:

    bcc="joe@site.com"

    bcc=('joe@site.com' "jane@home.net")

    bcc="Joe Smith <joe@site.com>"

FROM='from-address'
specifies the e-mail address of the author of the message that is being sent. The default value for FROM= is the e-mail address of the user who is running SAS. Specify this option, for example, when the person who is sending the message from SAS is not the author. You must enclose an address in quotation marks. You can specify only one e-mail address. To specify the author's real name along with the address, enclose the address in angle brackets (<>). Here are examples:

    from='martin@home.com'

    from="Brad Martin <martin@home.com>"

REPLYTO='replyto-address'
specifies the e-mail address(es) for who will receive replies. You must enclose an address in quotation marks. To specify more than one address, enclose the group of addresses in parentheses, enclose each address in quotation marks, and separate each address with a space. To specify a real name along with an address, enclose the address in angle brackets (<>). Here are examples:

    replyto='hiroshi@home.com'

    replyto=('hiroshi@home.com' 'akiko@site.com')

    replyto="Hiroshi Mori <mori@site.com>"
SUBJECT=subject
 specifies the subject of the message. If the subject contains special characters or
 more than one word (that is, it contains at least one blank space), you must
 enclose the text in quotation marks. Here are examples:

 subject=Sales

 subject="June Sales Report"

 Note: If you do not enclose a one-word subject in quotation marks, it is
 converted to uppercase.

 ATTACH='filename.ext' | ATTACH=('filename.ext' attachment-options)
 specifies the physical name of the file(s) to be attached to the message and any
 options to modify attachment specifications. The physical name is the name that is
 recognized by the operating environment. Enclose the physical name in quotation
 marks. To attach more than one file, enclose the group of files in parentheses,
 enclose each file in quotation marks, and separate each with a space. Here are
 examples:

 attach="/u/userid/opinion.txt"

 attach=('C:\Status\June2001.txt' 'C:\Status\July2001.txt')

 attach="user.misc.pds(member)"

 The attachment-options include the following:

 CONTENT_TYPE='content/type'
 specifies the content type for the attached file. You must enclose the value in
 quotation marks. If you do not specify a content type, SAS tries to determine
 the correct content type based on the filename. For example, if you do not
 specify a content type, a filename of home.html is sent with a content type of
 text/html.

 Aliases: CT= and TYPE=

 Default: If SAS cannot determine a content type based on the filename and
 extension, the default value is text/plain.

 ENCODING='encoding-value'
 specifies the text encoding of the attachment that is read into SAS. You must
 enclose the value in quotation marks. For valid encoding values, see
 “Encoding Values in SAS Language Elements” in SAS National Language

 EXTENSION='extension'
 specifies a different file extension to be used for the specified attachment. You
 must enclose the value in quotation marks. This extension is used by the
 recipient's e-mail program for selecting the appropriate utility to use for
 displaying the attachment. For example, the following results in the
 attachment home.html being received as index.htm:

 attach=("home.html" name="index" ext="htm")

 Alias: EXT=

 NAME='filename'
 specifies a different name to be used for the specified attachment. You must
 enclose the value in quotation marks. For example, the following results in
 the attachment home.html being received as index.html:

 attach=('home.html" name="index")
OUTENCODING='encoding-value'
specifies the resulting text encoding for the attachment to be sent. You must enclose the value in quotation marks. For valid encoding values, see “Encoding Values in SAS Language Elements” in SAS National Language Support (NLS): User’s Guide.

**CAUTION:**
Do not specify EBCDIC encoding values, because the SMTP e-mail interface does not support EBCDIC.

### PUT Statement Syntax for EMAIL (SMTP) Access Method

In the DATA step, after using the FILE statement to define your e-mail fileref as the output destination, use PUT statements to define the body of the message. For example,

```sas
filename mymail email 'martin@site.com' subject='Sending Email';
data _null_;  
  file mymail;  
  put 'Hi';  
  put 'This message is sent from SAS...';  
run;
```

You can also use PUT statements to specify e-mail directives that override the attributes of your message (the e-mail options like TO=, CC=, SUBJECT=, CONTENT_TYPE=, ATTACH=), or to perform actions such as send, abort, or start a new message. Specify only one directive in each PUT statement; each PUT statement can contain only the text that is associated with the directive that it specifies.

The directives that change the attributes of a message are as follows:

- **!EM_TO! to-address**
  replaces the current primary recipient address(es). The directive must be enclosed in quotation marks. To specify more than one address, enclose the group of addresses in parentheses, enclose each address in quotation marks, and separate each address with a space. To specify a real name along with an address, enclose the address in angle brackets (< >). Here are examples:

  ```sas
  put '!em_to! joe@site.com';
  put '!em_to! ("joe@site.com" "jane@home.net")';
  put '!em_to! Joe Smith <joe@site.com>';  
  **Tip:** Specifying !EM_TO! overrides the 'address' argument and the TO= e-mail option.
  
- **!EM_CC! cc-address**
  replaces the current copied recipient address(es). The directive must be enclosed in quotation marks. To specify more than one address, enclose the group of addresses in parentheses, enclose each address in quotation marks, and separate each address with a space. To specify real names along with addresses, enclose the address in angle brackets (< >). Here are examples:

  ```sas
  put '!em_cc! joe@site.com';
  put '!em_cc! ("joe@site.com" "jane@home.com")';
  put '!em_cc! Joe Smith <joe@site.com>';  
  ```
'!EM_BCC! bcc-address'
replaces the current blind copied recipient address(es) with addresses. These recipients are not visible to those in the !EM_TO! or !EM_CC! addresses. If you want to specify more than one address, then you must enclose the group of addresses in parentheses, enclose each address in quotation marks, and separate each address with a space. To specify real names along with addresses, enclose the address in angle brackets (< >). Here are examples:

put '!em_bcc! joe@site.com';

put '!em_bcc! ("joe@site.com" "jane@home.net");

put '!em_bcc! Joe Smith <joe@site.com>';

'!EM_FROM! from-address'
replaces the current address of the author of the message being sent, which could be either the default or the one specified by the FROM= e-mail option. The directive must be enclosed in quotation marks. You can specify only one e-mail address. To specify the author's real name along with the address, enclose the address in angle brackets (< >). Here are examples:

put '!em_from! martin@home.com';

put '!em_from! Brad Martin <martin@home.com>';

'!EM_REPLYTO! replyto-address'
replaces the current address(es) of who will receive replies. The directive must be enclosed in quotation marks. To specify more than one address, enclose the group of addresses in parentheses, enclose each address in quotation marks, and separate each address with a space. To specify a real name along with an address, enclose the address in angle brackets (< >). Here are examples:

put '!em_replyto! hiroshi@home.com';

put '!em_replyto! ("hiroshi@home.com" "akiko@site.com");

put '!em_replyto! Hiroshi Mori <mori@site.com>';

'!EM_SUBJECT! subject'
replaces the current subject of the message. The directive must be enclosed in quotation marks. If the subject contains special characters or more than one word (that is, it contains at least one blank space), you must enclose the text in quotation marks. Here are examples:

put '!em_subject! Sales';

put '!em_subject! "June Sales Report"';

'!EM_ATTACH! \\filename.ext | ATTACH=('filename.ext' attachment-options)
replaces the physical name of the file(s) to be attached to the message and any options to modify attachment specifications. The physical name is the name that is recognized by the operating environment. The directive must be enclosed in quotation marks, and the physical name must be enclosed in quotation marks. To
attach more than one file, enclose the group of files in parentheses, enclose each file in quotation marks, and separate each with a space. Here are examples:

```
put '!em_attach! /u/userid/opinion.txt';
put '!em_attach! ("C:\Status\June2001.txt" "C:\Status\July2001.txt")';
put '!em_attach! user.misc.pds(member)';
```

The `attachment-options` include the following:

- **CONTENT_TYPE='content/type'**
  specifies the content type for the attached file. You must enclose the value in quotation marks. If you do not specify a content type, SAS tries to determine the correct content type based on the filename. For example, if you do not specify a content type, a filename of `home.html` is sent with a content type of `text/html`.

  **Aliases:** CT= and TYPE=

  **Default:** If SAS cannot determine a content type based on the filename and extension, the default value is text/plain.

- **ENCODING='encoding-value'**
  specifies the text encoding to use for the attachment as it is read into SAS. You must enclose the value in quotation marks. For valid encoding values, see “Encoding Values in SAS Language Elements” in *SAS National Language Support (NLS): User’s Guide*.

- **EXTENSION='extension'**
  specifies a different file extension to be used for the specified attachment. You must enclose the value in quotation marks. This extension is used by the recipient’s e-mail program for selecting the appropriate utility to use for displaying the attachment. For example, the following results in the attachment `home.html` being received as `index.htm`:

  ```
  put '!em_attach! ("home.html" name="index" ext="htm")';
  ```

  **Alias:** EXT=

  **Default:** TXT

- **NAME='filename'**
  specifies a different name to be used for the specified attachment. You must enclose the value in quotation marks. For example, the following results in the attachment `home.html` being received as `index.html`:

  ```
  put '!em_attach! ("home.html" name="index")';
  ```

- **OUTENCODING='encoding-value'**
  specifies the resulting text encoding for the attachment to be sent. You must enclose the value in quotation marks. For valid encoding values, see “Encoding Values in SAS Language Elements” in *SAS National Language Support (NLS): User’s Guide*.

  **CAUTION:**
  Do not specify EBCDIC encoding values, because the SMTP e-mail interface does not support EBCDIC.
Here are the directives that perform actions:

'!EM_SEND!'
sends the message with the current attributes. By default, SAS sends a message when the fileref is closed. The fileref closes when the next FILE statement is encountered or the DATA step ends. If you use this directive, SAS sends the message when it encounters the directive, and again at the end of the DATA step. This directive is useful for writing DATA step programs that conditionally send messages or use a loop to send multiple messages.

'!EM_ABORT!'
aborts the current message. You can use this directive to stop SAS from automatically sending the message at the end of the DATA step. By default, SAS sends a message for each FILE statement.

'!EM_NEWMSG!'
clears all attributes of the current message that were set using PUT statement directives.

Details

Overview You can send electronic mail programmatically from SAS using the EMAIL (SMTP) access method. To send e-mail to an SMTP server, you first specify the SMTP e-mail interface with the EMAILSYS system option, use the FILENAME statement to specify the EMAIL device type, then submit SAS statements in a DATA step or in SCL code. This has several advantages:

- You can use the logic of the DATA step or SCL to subset e-mail distribution based on a large data set of e-mail addresses.
- You can automatically send e-mail upon completion of a SAS program that you submitted for batch processing.
- You can direct output through e-mail based on the results of processing.

In general, DATA step or SCL code that sends e-mail has the following components:

- a FILENAME statement with the EMAIL device-type keyword
- e-mail options specified in the FILENAME or FILE statement that indicate e-mail recipients, subject, attached file(s), and so on
- PUT statements that define the body of the message
- PUT statements that specify e-mail directives (of the form !EM_directive!) that override the e-mail options (for example, TO=, CC=, SUBJECT=, ATTACH=) or perform actions such as send, abort, or start a new message.

Examples

Example 1: Sending E-mail with an Attachment Using a DATA Step In order to share a copy of your SAS configuration file with another user, you could send it by submitting the following program. The e-mail options are specified in the FILENAME statement:

```sas
filename mymail email "JBrown@site.com"
   subject="My SAS Configuration File"
   attach="/u/sas/sasv8.cfg";

data _null_; file mymail;
```
The following program sends a message and two file attachments to multiple recipients. For this example, the e-mail options are specified in the FILE statement instead of the FILENAME statement.

```sas
filename outbox email "ron@acme.com";

data _null_;
  file outbox
    to=("ron@acme.com" "humberto@acme.com")
    /* Overrides value in */
    /* filename statement */
    cc=("miguel@acme.com" "loren@acme.com")
    subject="My SAS Output"
    attach=("C:\sas\results.out" "C:\sas\code.sas")
  ;
  put 'Folks,'
    put 'Attached is my output from the SAS,'
    put 'program I ran last night,'
    put 'It worked great!';
run;
```

**Example 2: Using Conditional Logic in a DATA Step**

You can use conditional logic in a DATA step in order to send multiple messages and control which recipients get which message. For example, in order to send customized reports to members of two different departments, the following program produces an e-mail message and attachments that are dependent on the department to which the recipient belongs. In the program, the following occurs:

1. In the first PUT statement, the !EM_TO! directive assigns the TO attribute.
2. The second PUT statement assigns the SUBJECT attribute using the !EM_SUBJECT! directive.
3. The !EM_SEND! directive sends the message.
4. The !EM_NEWMSG! directive clears the message attributes, which must be used to clear message attributes between recipients.
5. The !EM_ABORT! directive aborts the message before the RUN statement causes it to be sent again. The !EM_ABORT! directive prevents the message from being automatically sent at the end of the DATA step.

```sas
filename reports email "Jim.Smith@work.com";

data _null_;
  file reports;
    length name dept $ 21;
    input name dept;
    put '!EM_TO! ' name;
    put '!EM_SUBJECT! Report for ' dept;
    put name ',';
    put 'Here is the latest report for ' dept '.
    if dept='marketing' then
```
Example 3: Sending Procedure Output in E-mail  You can use e-mail to send procedure output. This example illustrates how to send ODS HTML in the body of an e-mail message. Note that ODS HTML procedure output must be sent with the RECORD_SEPARATOR (RS) option set to NONE.

```sas
filename outbox email
to='susan@site.com'
type='text/html'
subject='Temperature Conversions';

data temperatures;
do centigrade = -40 to 100 by 10;
fahrenheit = centigrade*9/5+32;
output;
end;
run;

ods html
body=outbox /* Mail it! */
rs=none;

title 'Centigrade to Fahrenheit Conversion Table';

proc print;
id centigrade;
var fahrenheit;
run;

ods html close;
```

Example 4: Creating and E-mailing an Image  The following example illustrates how to create a GIF image and send it from SAS in an e-mail message:

```sas
filename gsasfile email
to='Jim@acme.com'
type='image/gif'
subject="SAS/GRAPH Output";

goptions dev=gif gsfname=gsasfile;
```
FILENAME Statement, FTP Access Method

Enables you to access remote files by 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 from 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 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.
If the file has an IBM 370 format and a record format of FB or FBA, and if the
ENCODING= option is specified, then you must also specify the LRECL= option. If
the length of a record is shorter than the value of LRECL, then SAS pads the record
with blanks until the record length is equal to the value of LRECL.
**Operating Environment Information:** For details about 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, then you do not need to specify a filename. Instead, put empty quotation marks in the statement. See Example 1 on page 1198.

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

**Tip:** If you use the DIR option, specify the directory in this argument.

**ftp-options**
specifies details that are specific to your operating environment such as file attributes and processing attributes.

**Operating Environment Information:** For more information on some of these FTP options, see the SAS documentation for your operating environment.

**FTP Options**

**BINARY**
is fixed-record format. Thus, all records are of size LRECL with no line delimiters. Data is transferred in image (binary) mode.

The BINARY option overrides the value of RECFM= in the FILENAME FTP statement, if specified, and forces a binary transfer.

**Alias:** RECFM=F

**Interaction:** If you specify the BINARY option and the S370V or S370VS option, then SAS ignores the BINARY option.

**BLOCKSIZE=** blocksize

where blocksize is the size of the data buffer in bytes.

**Default:** 32768

**CD='directory'**
issues a command that changes the working directory for the file transfer to the directory that you specify.

**Interaction:** The CD and DIR options are mutually exclusive. If both are specified, FTP ignores the CD option and SAS writes an informational note to the log.

**DEBUG**
writes to the SAS log informational messages that are sent to and received from the FTP server.

**DIR**
enables you to access directory files or PDS/PDSE members. Specify the directory name in the external-file argument. You must use valid directory syntax for the specified host.

**Tip:** If you want FTP to create the directory, then use the NEW option in conjunction with the DIR option. The NEW option will be ignored if the directory exists.

**Tip:** If the NEW option is omitted and you specify an invalid directory, then a new directory will not be created and you will receive an error message.

**Tip:** The maximum number of directory or z/OS PDSE members that can be open simultaneously is limited by the number of sockets that can be open simultaneously on an FTP server. The number of sockets that can be open
simultaneously is proportional to the number of connections that are set up during the installation of the FTP server. You might want to limit the number of sockets that are open simultaneously to avoid performance degradation.

**Interaction:** The CD and DIR options are mutually exclusive. If both are specified, FTP ignores the CD option and SAS writes an informational note to the log.

**Featured in:** Example 10 on page 1201

ENCODING=encoding-value

specifies the encoding to use when reading from or writing to the external file. The value for ENCODING= indicates that the external file has a different encoding from the current session encoding.

When you read data from an external file, SAS transcodes the data from the specified encoding to the session encoding. When you write data to an external file, SAS transcodes the data from the session encoding to the specified encoding.


**Default:** SAS assumes that an external file is in the same encoding as the session encoding.

**Tip:** The data is transferred in image or binary format and is in local data format. Thus, you must use appropriate SAS informats to read the data correctly.

HOST='host'

where host is the network name of the remote host with the FTP server running.

You can specify either the name of the host (for example, server.pc.mydomain.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 of the data.

**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 LSFILE= option in addition to LS.

LSFILE='character-string'

in combination with the LS option, specifies a character string that enables 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 of the files that start with **sales** and end with **sas**:

```
filename myfile ftp '' ls lsize='sales*.sas'
other-ftp-options;
```

**MGET**

transfers multiple files, similar to the FTP command MGET.

**Tip:** The whole transfer is treated as 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.

**Restriction:** The MPROMPT option is not available on z/OS for batch processing.

**NEW**

specifies that you want FTP to create the directory when you use the DIR option.

**Tip:** The NEW option will be ignored if the directory exists.

**Restriction:** The NEW option is not available under z/OS.

**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**, then the remote host might require that you specify your e-mail address as the password.

**Tip:** To use an encoded password, use the PWENCODE procedure in order to disguise the text string, and then enter the encoded password for the PASS= option. For more information, see “The PWENCODE Procedure” in *Base SAS Procedures Guide*.

**Featured in:** Example 6 on page 1200

**PORT=portno**

where **portno** is the port that the FTP daemon monitors on the respective host.

The **portno** can be any number between 0 and 65535 that uniquely identifies a service.

**Tip:** In the Internet community, there is a list of predefined port numbers for specific services. For example, the default port for FTP is 21. A partial list of port numbers is usually available in the `/etc/services` file on any UNIX computer.

**PROMPT**

specifies to prompt for the user login password, if necessary.

**Restriction:** The PROMPT option is not available for batch processing under z/OS.

**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 FTP 'SITE' or 'service' command to send to the FTP server.

FTP servers use SITE commands to provide services that are specific to a system and are essential to file transfer but not common enough to be included in the protocol.
For instance, `rcmd='site rdw'` preserves the record descriptor word (RDW) of a z/OS variable blocked data set as a part of the data. See S370V and S370VS below.

**Interaction:** Some FTP service commands might not run at a particular client site depending on the security permissions and the availability of the commands.

**Tip:** You can specify multiple FTP service commands if you separate them by semicolons.

**Examples:**

```plaintext
rcmd='ascii;site unmask 002'

rcmd='stat;site chmod 0400 -aclin/abc.txt'
```

**RECFM=rcfm**

where `rcfm` is one of three record formats:

- **F** is fixed-record format. Thus, all records are of size LRECL with no line delimiters. Data is transferred in image (binary) mode.

  **Alias:** BINARY

  The BINARY option overrides the value of RECFM= in the FILENAME FTP statement, if specified, and forces a binary transfer.

- **S** is stream-record format. Data is transferred in image (binary) mode.

  **Interaction:** The amount of data that is read is controlled by the current LRECL value or by the value of the NBYTE= variable in the 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 1229 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.

  **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, then you might want to use the S370V or S370VS options instead of the RECFM= option. See S370V and S370VS below.

**Default:** V

**Interaction:** If you specify the RECFM= option and the S370V or S370VS option, then SAS ignores the RECFM= option.

**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.

**Interaction:** If you specify this option and the RECFM= option, then SAS ignores the RECFM= option.
Tip: The data is transferred in image or binary format and is in local data format. Thus, you must use appropriate SAS informats to read the data correctly on non-EBCDIC hosts.

Tip: Use the rcmd='site rdw' option when you transfer a z/OS data set with a variable-record format to another z/OS data set with a variable-record format to preserve the record descriptor word (rdw) of each record. By default, most FTP servers remove the rdw that exists in each record before it is transferred.

Typically, the 'SITE RDW' command is not necessary when you transfer a data set with a z/OS variable-record format to ASCII, or when you transfer an ASCII file to a z/OS variable-record format.

S370VS indicates that the file that is being read is in IBM 370 variable-spanned format.

Interaction: If you specify this option and the RECFM= option, then SAS ignores the RECFM= option.

Tip: The data is transferred in image or binary format and is in local data format. Thus, you must use appropriate SAS informats to read the data correctly on non-EBCDIC hosts.

Tip: Use the rcmd='site rdw' option when you transfer a z/OS data set with a variable-record format to another z/OS data set with a variable-record format to preserve the record descriptor word (rdw) of each record. By default, most FTP servers remove the rdw that exists in each record before it is transferred.

Typically, the 'SITE RDW' command is not necessary when you transfer a data set with a z/OS variable-record format to ASCII, or when you transfer an ASCII file to a z/OS variable-record format.

USER='username'

where username is used to log in to the FTP server.

Restriction: The USER option is not available under z/OS for batch processing.

Interaction: If PROMPT is specified, but USER= is not, then 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 smythe, and prompts smythe for a password:

    filename dir ftp '' ls user='smythe'
    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 quotation marks are required by the syntax, however, you must include them.  △
Example 2: Reading a File from a Remote Host  This example reads a file called sales in the directory /u/kudzu/mydata from the remote UNIX host hp720:

```
filename myfile ftp 'sales' 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 the \remote directory of the ftp server for the user bbailey on the host winnt.pc:

```
filename create ftp 'test.dat' cd='\remote'
  host='winnt.pc'
  user='bbailey' prompt recfm=v;

data _null_;  /* Create a view */
  file create;
  do i=1 to 10;
    put i=;
  end;
run;
```

Example 4: Reading an S370V-Format File on z/OS  This example reads an S370V-format file from a z/OS system. See the RCMD= on page 1196 option for more information about RCMD='site rdw'.

```
filename viewdata ftp 'sluggo.stat.data'
  user='sluggo' host='zoshost1'
  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 log in 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 1196 under “FTP Options.”
```

```
filename anon ftp '' 1s host='130.96.6.1'
  user='anonymous';

data _null_;  /* Create a view */
  infile anon;
```


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: Using an Encoded Password  This example shows you how to use an encoded password in the FILENAME statement.

In a separate SAS session, use the PWENCODE procedure to encode your password and make note of the output:

```sas
proc pwencode in='MyPass1';
runc;
```

The following output appears in the SAS log:

```
(sas001)TX1QYXNzMQ==
```

You can now use the entire encoded password string in your batch program.

```sas
filename myfile ftp 'sales' cd='/u/kudzu/mydata'
    user='tjbarry' host='hp720.hp.mycompany.com'
    pass='''(sas001)TX1QYXMZ=='';
```

Example 7: Importing a Transport Data Set

This example uses the CIMPORT procedure to import a transport data set from a host named myshost1 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 fully qualified data set name, then use double quotation marks and single quotation marks. Otherwise, the system will append the profile prefix to the name that you specify.

```sas
%let user=calvin;
%let pw=xxxxx;
filename inp ftp 'calvin.mat1.cpo' user="&user"
    pass="&pw" rcmd='binary'
    host='mvshost1';
```

```sas
proc cimport library=sasuser infile=inp;
runc;
```

Example 8: 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.

```sas
filename inp ftp 'mat64.cpo' user='calvin'
    pass="xxxxx" host='mvshost1'
    lrecl=80 recfm=f blocksize=8000
    rcmd='site blocksize=800 recfm=fb lrecl=80';
```

```sas
proc cport library=mylib file=inp;
runc;
```

Example 9: 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 XPORT engine. The PROC COPY step copies all data sets from the SAS data library that are referenced by MYLIB to the XPORT engine. Output from the PROC CONTENTS step confirms that the copy was successful:

```plaintext
filename inp ftp 'mat65.cpo' user='calvin'
   pass="xxxx" host='mvhost1'
   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;
```

**Example 10: Reading and Writing from Directories**  This example reads the file `ftpmem1` from a directory on a UNIX host and writes the file `ftpout1` to a different directory on another UNIX host.

```plaintext
filename indir ftp '/usr/proj2/dir1' DIR
   host="host1.mycompany.com"
   user="xxxx" prompt;
filename outdir ftp '/usr/proj2/dir2' DIR
   host="host2.mycompany.com"
   user="xxxx" prompt;

data _null_; infile indir(ftpmem1) truncover;
   input;
   file outdir(ftpout1);
   put _infile_;
```

The file `ftpout1` is written to `/usr/proj2/dir2/ftpout1`.

**See Also**

Statements:
- “FILENAME Statement” on page 1172
- “FILENAME Statement, CATALOG Access Method” on page 1178
- “FILENAME Statement, EMAIL (SMTP) Access Method” on page 1183
- “FILENAME Statement, SOCKET Access Method” on page 1201
- “FILENAME Statement, URL Access Method” on page 1205
- “LIBNAME Statement” on page 1286

**FILENAME Statement, SOCKET Access Method**

Enables you to read from or write to a TCP/IP socket
Valid: anywhere
Category: Data Access

Syntax

1. `FILENAME fileref SOCKET 'hostname:portno'
   <tcpip-options>;

2. `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 1203 under `TCPIP-Options`.

**TCPIP-Options**

`BLOCKSIZE=blocksize`

where blocksize is the size of the socket data buffer in bytes.

**Default:** 8192

`ENCODING=encoding-value`

specifies the encoding to use when reading from or writing to the socket. The value for ENCODING= indicates that the socket has a different encoding from the current session encoding.

When you read data from a socket, SAS transcodes the data from the specified encoding to the session encoding. When you write data to a socket, SAS transcodes the data from the session encoding to the specified encoding.

LRECL=\texttt{lrecl}
where \texttt{lrecl} is the logical record length.

\textbf{Default:} 256

RECFM=\texttt{rcfm}
where \texttt{rcfm} is one of three record formats:

- \texttt{F}\hspace{1em}is fixed record format. Thus, all records are of size LRECL with no line delimiters. Data are transferred in image (binary) mode.

- \texttt{S}\hspace{1em}is stream record format.

\textbf{Tip:} Data are transferred in image (binary) mode.

\textbf{Interactions:} The amount of data that is read is controlled by the current LRECL value or the value of the NBYTE= variable in the 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.

\textbf{See Also:} The NBYTE= option on page 1229 in the INFILE statement.

- \texttt{V}\hspace{1em}is variable record format (the default).

\textbf{Tip:} In this format, records have varying lengths, and they are transferred in text (stream) mode.

\textbf{Tip:} Any record larger than LRECL is truncated.

\textbf{Default:} V

RECONN=\texttt{conn-limit}
where \texttt{conn-limit} is the maximum number of connections that the server will accept.

\textbf{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 \texttt{conn-limit} has been reached.

TERMSTR='\texttt{eol-char}'
where \texttt{eol-char} is the line delimiter to use when RECFM=V. There are three valid values:

- \texttt{CRLF}\hspace{1em}carriage return (CR) followed by line feed (LF).

- \texttt{LF}\hspace{1em}line feed only (the default).

- \texttt{NULL}\hspace{1em}NULL character (0x00).

\textbf{Default:} LF

\textbf{Restriction:} Use this option only when RECFM=V.

\textbf{Details}

\textbf{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 Technical Support (free of charge) that allows a Visual Basic application to communicate through the sockets.

Using 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.

Using 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, SAS 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:

```plaintext
filename local socket ':5000' server reconn=3;
/*The server is using a reserved */
```
FILENAME Statement, URL Access Method

Enables you to access remote files by using the URL access method

Valid: anywhere

Category: Data Access

Syntax

FILENAME fileref URL 'external-file'
<url-options>;

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 Statement” on page 1172
  “FILENAME Statement, CATALOG Access Method” on page 1178
  “FILENAME Statement, EMAIL (SMTP) Access Method” on page 1183
  “FILENAME Statement, FTP Access Method” on page 1193
  “FILENAME Statement, URL Access Method” on page 1205
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 that you want to read from on a URL server. The access method must be in one of these forms:

- `http://hostname/file`
- `https://hostname/file`
- `http://hostname:portno/file`
- `https://hostname:portno/file`

**Operating Environment Information:** For details about 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 Uniform 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 is transferred in image (binary) mode.

S     is stream-record format. Data is 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 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 1229 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 on 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

The Secure Sockets Layer (SSL) protocol is used when the URL begins with “https” instead of “http”. The SSL protocol provides network security and privacy. Developed by Netscape Communications, SSL uses encryption algorithms that include RC2, RC4, DES, tripleDES, IDEA, and MD5. Not limited to providing only encryption services, SSL can also perform client and server authentication and use message authentication codes. SSL is supported by both Netscape Navigator and Internet Explorer. Many Web sites use the protocol to provide confidential user information such as credit card numbers. The SSL protocol is application independent, enabling protocols such as HTTP, FTP, and Telnet to be layered transparently above it. SSL is optimized for HTTP.

Operating Environment Information: Using the FILENAME statement requires information that is specific to your operating environment. The URL access method is fully documented here, but for more information about how to specify filenames, 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:

```plaintext
c_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:

```plaintext
c_filename foo url
  'http://support.sas.com/techsup/service_intro.html';

data _null_
  infile foo length=len;
  input record $varying200. len;
  put record $varying200. len;
  if _n_=15 then stop;
run;
```

See Also

Statements:

- “FILENAME Statement” on page 1172
- “FILENAME Statement, CATALOG Access Method” on page 1178
- “FILENAME Statement, EMAIL (SMTP) Access Method” on page 1183
- “FILENAME Statement, FTP Access Method” on page 1193
- “FILENAME Statement, SOCKET Access Method” on page 1201
- “Using SSL in UNIX Environments” in SAS Companion for UNIX Environments
- “Using SSL under Windows” in SAS Companion for Windows

---

FOOTNOTE Statement

Writes up to 10 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.

See: FOOTNOTE Statement in the documentation for your operating environment.

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 enclose text within quotation marks.

Details

A FOOTNOTE statement takes effect when the step or RUN group with which it is associated executes. After 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 that is 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.

You can modify footnotes with the Output Delivery System. See Example 3 on page 1412.

Examples

These examples of a FOOTNOTE statement result in the same footnote:

- `footnote8 "Managers' Meeting";`
- `footnote8 ‘Managers’ Meeting’;

See Also

Statement:

“TITLE Statement” on page 1409

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

Syntax

```
FORMAT variable-1 < . . . variable-n> <format> <DEFAULT=default-format>;
FORMAT variable-1 < . . . variable-n> format <DEFAULT=default-format>;
FORMAT variable-1 < . . . variable-n> format variable-1 < . . . variable-n> 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 3 on page 1212. 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 modifier, see “PUT Statement, List” on page 1366.

**See also:** “Formats by Category” on page 81

`DEFAULT=default-format`

specifies a temporary default format for displaying the values of variables that are not 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 BESTw. as the default numeric format and $w. as the default character format.

**Restriction:** Use this option only in a DATA step.

**Tip:** A DEFAULT= specification can occur anywhere in a FORMAT statement. It can specify either a numeric default, a character default, or both.

**Featured in:** Example 1 on page 1211

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 Base 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.

```sas
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.

<table>
<thead>
<tr>
<th>#</th>
<th>Variable</th>
<th>Type</th>
<th>Len</th>
<th>Pos</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>W</td>
<td>Char</td>
<td>3</td>
<td>16</td>
<td>$CHAR3.</td>
</tr>
<tr>
<td>3</td>
<td>X</td>
<td>Num</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Y</td>
<td>Num</td>
<td>8</td>
<td>0</td>
<td>10.3</td>
</tr>
<tr>
<td>4</td>
<td>Z</td>
<td>Char</td>
<td>11</td>
<td>19</td>
<td></td>
</tr>
</tbody>
</table>
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: Associating Multiple Variables with a Single Format

This example uses the FORMAT statement to assign a single format to multiple variables.

```sas
data report;
  input Item $ 1--6 Material $ 8--14 Investment 16--22 Profit 24--31;
  format Item Material $upcase9. Investment Profit dollar15.2;
  datalines;
  shirts cotton 2256354 83952175
  ties silk 498678 2349615
  suits silk 9482146 69839563
  belts leather 7693 14893
  shoes leather 7936712 22964;
run;
```

Output 7.4 Results from Associating Multiple Variables with a Single Format

```
<table>
<thead>
<tr>
<th>Obs</th>
<th>Item</th>
<th>Material</th>
<th>Investment</th>
<th>Profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SHIRTS</td>
<td>COTTON</td>
<td>$2,256,354.00</td>
<td>$83,952,175.00</td>
</tr>
<tr>
<td>2</td>
<td>TIES</td>
<td>SILK</td>
<td>$498,678.00</td>
<td>$2,349,615.00</td>
</tr>
<tr>
<td>3</td>
<td>SUITS</td>
<td>SILK</td>
<td>$9,482,146.00</td>
<td>$69,839,563.00</td>
</tr>
<tr>
<td>4</td>
<td>BELTS</td>
<td>LEATHER</td>
<td>$7,693.00</td>
<td>$14,893.00</td>
</tr>
<tr>
<td>5</td>
<td>SHOES</td>
<td>LEATHER</td>
<td>$7,936,712.00</td>
<td>$22,964.00</td>
</tr>
</tbody>
</table>
```

Example 3: 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.

```sas
data rtest;
set rtest;
```
GO TO Statement

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

```lisp
format x;
run;
```

See Also

Statement:

“ATTRIB Statement” on page 1111
“The DATASETS Procedure” in Base SAS Procedures Guide
is false, SAS executes the PUT statement and the statement that is associated with the GO TO label:

```sas
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.

```sas
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 Statement” on page 1145
- “Labels, Statement” on page 1281
- “LINK Statement” on page 1296
- “RETURN Statement” on page 1387

---

**IF Statement, 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:
  ```sas
  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:
  ```sas
  if age;
  ```

See Also

Data Set Options:
- “WHERE= Data Set Option” on page 61

Statements:
- “DELETE Statement” on page 1140
- “IF-THEN/ELSE Statement” on page 1216
- “WHERE Statement” on page 1421
- “WHERE-Expression Processing” in SAS Language Reference: Concepts

IF-THEN/ELSE Statement

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 non-zero, zero, or missing. A non-zero 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.

```sas
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=.;
    '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...
```

...
%INCLUDE Statement

Brings a SAS programming statement, data lines, or both, into a current SAS program

Valid: anywhere
Category: Program Control
Alias: %INC
See: %INCLUDE Statement in the documentation for your operating environment.

Syntax

```plaintext
%INCLUDE source(s)
</<SOURCE2> <S2=length> <operating-environment-options>>;
```

Arguments

**source**

describes the location of the information that you want to access with the %INCLUDE statement. There are three possible sources:

<table>
<thead>
<tr>
<th>Source</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>file-specification</td>
<td>specifies an external file</td>
</tr>
<tr>
<td>internal-lines</td>
<td>specifies lines that are entered earlier in the same SAS job or session</td>
</tr>
<tr>
<td>keyboard-entry</td>
<td>specifies statements or data lines that you enter directly from the terminal</td>
</tr>
</tbody>
</table>

**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. By default, the SPOOL system option is set to NOSPOOL. The SPOOL system option must be in effect in order to use %INCLUDE statements with internal line references. 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 “S= System Option” on page 1594 and “S2= System Option” on page 1596.

**operating-environment-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:

```sas
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:

```sas
%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:

```sas
filename in1 'MYFILE';
```
You can later access MYFILE with the fileref IN1:

```sas
%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:

```sas
filename storage
   'aggregate-storage-location';
```

You can later include a file using this statement:

```sas
%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:

```sas
%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.

```sas
%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:

```sas
%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:

```sas
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

```sas
where month= 'January';
```
title 'Data for month of January';

After you enter statements, you can use %RUN to resume processing by typing

```sas
%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 MYLIBINCLUDE. When no entry type is specified, the default is CATAMS.

```sas
filename dir catalog 'mylib.include';
%include dir(mem1);
%include dir(mem2);
%include dir(mem3);
```

### See Also

Statements:
- “%LIST Statement” on page 1300
- “%RUN Statement” on page 1390

---

**INFILE Statement**

Identifies an external file to read with an INPUT statement

Valid: in a DATA Step

Category: File-handling

Type: Executable

See: INFILE Statement in the documentation for your operating environment.

### Syntax

```
INFILE file-specification <options> <operating-environment-options>;
INFILE DBMS-specifications;
```

### Arguments

**file-specification**

identifies the source of the input data records, which is an external file or instream 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 Statement” on page 1172

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 about 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 Statement” on page 1172

CARD$ | 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 1236

**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 1228

**Featured in:** Example 8 on page 1241

**DELIMITER= delimiter(s)**
specifies an alternate delimiter (other than a blank) to be used 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.
**Infile Statement**  
**Chapter 7**

**Featured in:** Example 1 on page 1236  

*character-variable*  
specifies a character variable whose value becomes the delimiter.

**Alias:** DLM=  
**Default:** Blank space  

**See:** “Reading Delimited Data” on page 1234  
**See Also:** DSD (delimiter-sensitive data) option on page 1226  
**Featured in:** Example 1 on page 1236

DSD (delimiter-sensitive data)  
specifies that when data values are enclosed in quotation marks, delimiters within the value be treated as character data. The DSD option changes how SAS treats delimiters when you use LIST input 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 string that is enclosed in quotation marks. 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 1234  
**See Also:** DELIMITER= on page 1225  
**Featured in:** Example 1 on page 1236 and Example 2 on page 1237

ENCODING= ’encoding-value’  
specifies the encoding to use when reading from the external file. The value for ENCODING= indicates that the external file has a different encoding from the current session encoding.

When you read data from an external file, SAS transcodes the data from the specified encoding to the session encoding.


**Default:** SAS assumes that an external file is in the same encoding as the session encoding.  
**Featured in:** Example 11 on page 1244

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 1226 when END= is invalid.  
**Featured in:** Example 5 on page 1239

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 1226, EOV= on page 1227, and UNBUFFERED on page 1231

**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 1226 and EOF= on page 1226

**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 contains 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 1227

**Featured in:** Example 5 on page 1239

**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 1233

**Featured in:** Example 5 on page 1239

**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 1235

See Also:  MISSOVER on page 1229, STOPOVER on page 1231, and TRUNCOVER on page 1231

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 965 is useful when the field width varies.

Featured in:  Example 4 on page 1239 and Example 7 on page 1240

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 1225 and N= on page 1229

Featured in:  Example 8 on page 1241

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, then 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, then 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 file characteristics of your operating environment

**Restriction**: LRECL is not valid when you use the DATALINES file specification.

**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) might be missing and you want SAS to
assign missing values to the corresponding variable.

**See**: “Reading Past the End of a Line” on page 1235

**See Also**: FLOWOVER on page 1228, SCANOVER on page 1230, STOPOVER on
page 1231, and TRUNCOVER on page 1231

**Featured in**: Example 2 on page 1237

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, then 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. Here is an 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 is
read from that record.

**Featured in**: Example 8 on page 1241

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, then the FTP and
SOCKET access methods return the number of bytes that are currently
available in the input buffer.

**See**: The RECFM= option on page 1203 in the FILENAME statement, SOCKET
access method, and the RECFM= option on page 1197 in the FILENAME
statement, FTP access method.
OBS=record-number | MAX

record-number specifies the record number of the last record to read in an input file that is read sequentially.

MAX specifies the maximum number of observations to process, which will be at least as large as the largest signed, 32–bit integer. The absolute maximum depends on your host operating environment.

Default: MAX

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 1229

PRINT | NOPRINT

specifies whether the input file contains carriage-control characters.

Tip: To read a 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 the 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 1235

See Also: FLOWOVER on page 1228, MISSOVER on page 1229, STOPOVER on page 1231, and TRUNCOVER on page 1231

Featured in: Example 3 on page 1238

SHAREBUFFERS

specifies that the FILE statement and the INFILE statement share the same buffer.

Alias: SHAREBUFS

CAUTION:

When using SHAREBUFFERS, RECFM=V, and _INFILE_, use caution if you read a record with one length and update the file with a record of a different length. The
length of the record can change by modifying _INFILE_. One option to avoid this potential problem is to pad or truncate _INFILE_ so that the original record length is maintained. ▲

**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 6 on page 1240

**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 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 1235

**See Also:** FLOWOVER on page 1228, MISSOVER on page 1229, SCANOVER on page 1230, and TRUNCOVER on page 1231

**Featured in:** Example 2 on page 1237

**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 1235

**See Also:** FLOWOVER on page 1228, MISSOVER on page 1229, SCANOVER on page 1230, and STOPOVER on page 1231

**Featured in:** Example 3 on page 1238

**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 instream 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. Ensure that the _INFILE_= specification is the first occurrence 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_.

**Tip:** The _INFILE_ variable does not have a fixed width. When you assign a value to the _INFILE_ variable, the length of the variable changes to the length of the value that is assigned.

**Main Discussion:** “Accessing the Contents of the Input Buffer” on page 1233

**Featured in:** Example 9 on page 1241 and Example 10 on page 1243

**Operating Environment 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 enables 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 is 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. See “Reading Long Instream Data Records” on page 1235 for more information.
Statements △ INFILE Statement

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 5 on page 1239.

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 6 on page 1240.

To update individual fields within a record instead of the entire record, see the term SHAREBUFFERS under “Arguments” on page 1224.

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 variables, _INFILE_ is not written to the data set.

When you specify the _INFILE_= option in an 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, then the default length of the new variable is 32,767. You cannot 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, the following PUT statement writes the contents of the input buffer by using a hexadecimal format.

```
put _infile_ $hex100.;
```

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 an INFILE statement, even when you use the N= option to specify multiple buffers. You can access
all the N= buffers, but you must use an 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 delimiter-sensitive data (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 is 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.

```sas
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:

<table>
<thead>
<tr>
<th>OBS</th>
<th>TEST1</th>
<th>TEST2</th>
<th>TEST3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>91</td>
<td>87</td>
<td>95</td>
</tr>
<tr>
<td>2</td>
<td>97</td>
<td>.</td>
<td>92</td>
</tr>
</tbody>
</table>

To correct the problem, use the DSD option in the INFILE statement.

```sas
infile datalines dsd;
```

Now the INPUT statement detects the two consecutive delimiters and therefore assigns a missing value to variable TEST2 in the second observation.

<table>
<thead>
<tr>
<th>OBS</th>
<th>TEST1</th>
<th>TEST2</th>
<th>TEST3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>91</td>
<td>87</td>
<td>95</td>
</tr>
<tr>
<td>2</td>
<td>97</td>
<td>.</td>
<td>92</td>
</tr>
<tr>
<td>3</td>
<td>.</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The DSD option also enables list input to read a character value that contains a delimiter within a quoted string. For example, if data is 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 1236.

**Reading Long Instream Data Records** You can use the INFILE statement with the DATALINES file specification to process instream data. An INPUT statement reads the data records that follow the DATALINES statement. If you use the CARDIMAGE system option, or if this option is the default for your system, then SAS processes the data lines exactly like 80-byte punched card images that are padded with blanks. The default FLOWOVER option in the INFILE statement causes the INPUT statement to read the next record if it does not find values in the current record for all of the variables in the statement. To ensure that your data is processed correctly, use an external file for input when record lengths are greater than 80 bytes.

*Note:* The NOCARDIMAGE system option (see “CARDIMAGE System Option” on page 1489) specifies that data lines not be treated as if they were 80-byte card images. The end of a data line is always treated as the end of the last token, except for strings that are enclosed in quotation marks.

**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, then 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 statement reached past the end of a line.
```

Several options are available to change the INPUT statement behavior when an end of line is reached. 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, used with '@character-string' scans the input record until it finds the specified character-string. The FLOWOVER option restores the default behavior.

The TRUNCOVER and MISSOVER options are similar. Both options set the remaining INPUT statement variables to missing values. 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 that is 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 this data to create a SAS data set. Only one of the input records is as long as the informatted length of the variable TESTNUM.

```sas
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, then 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 7.5 on page 1236 to compare the SAS data sets.

Table 7.5  The Value of TESTNUM Using Different INFILE Statement Options

<table>
<thead>
<tr>
<th>OBS</th>
<th>FLOWOVER</th>
<th>MISSOVER</th>
<th>TRUNCOVER</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>22</td>
<td>.</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4444</td>
<td>.</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td>55555</td>
<td>.</td>
<td>333</td>
</tr>
<tr>
<td>4</td>
<td>.</td>
<td>4444</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>55555</td>
<td>55555</td>
<td></td>
</tr>
</tbody>
</table>

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 follows in the job stream. You can use the INFILE statement with the file specification DATALINES to take advantage of certain data-reading options that affect how the INPUT statement reads instream 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 instream 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, then 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
  ?a8b9
; 

The output that PROC PRINT generates shows the resulting NUMS data set. Values are missing for variables in the first and second observations 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 7.5  The NUMS Data Set

<table>
<thead>
<tr>
<th>OBS</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>.</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>5</td>
<td>.</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

This DATA step uses modified list input and the DSD option to read data that is separated by commas and that might contain commas as part of a character value:

data scores;
  infile datalines dsd;
  input Name : $9. Score
    Team : $25. Div $;
  datalines;
  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.

Output 7.6  Data Set SCORES

<table>
<thead>
<tr>
<th>OBS</th>
<th>NAME</th>
<th>SCORE</th>
<th>TEAM</th>
<th>DIV</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Joseph</td>
<td>76</td>
<td>Red Racers, Washington</td>
<td>AAA</td>
</tr>
<tr>
<td>2</td>
<td>Mitchel</td>
<td>82</td>
<td>Blue Bunnies, Richmond</td>
<td>AAA</td>
</tr>
<tr>
<td>3</td>
<td>Sue Ellen</td>
<td>74</td>
<td>Green Gazelles, Atlanta</td>
<td>AA</td>
</tr>
</tbody>
</table>

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 five temperature values. Use the MISSOVER option so that these values are set to missing.

```sas
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 the 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:

```sas
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: Scanning Variable-Length Records for a Specific Character String**

This example shows how to use TRUNCOVER in combination with SCANOVER to pull phone numbers from a phone book. The phone number is always preceded by the word “phone:”. Because the phone numbers include international numbers, the maximum length is 32 characters.

```sas
filename phonebk host-specific-path;
data _null_; file phonebk;
  input line $80.;
  put line;
  datalines;
  Jenny's Phone Book
  Jim Johanson phone: 619-555-9340
  Jim wants a scarf for the holidays.
  Jane Jovalley phone: (213) 555-4820
  Jane started growing cabbage in her garden.
  Her dog's name is Juniper.
  J.R. Hauptman phone: (49)12 34-56 78-90
  J.R. is my brother.
;
run;
```

Use @’phone:’ to scan the lines of the file for a phone number and position the file pointer where the phone number begins. Use TRUNCOVER in combination with
SCANOVER to skip the lines that do not contain 'phone:' and write only the phone numbers to the log.

data _null_
  ;
  infile phonebk truncover scanover;
  input @‘phone:’ phone $32.;
  put phone=;
run;

The program writes the following lines to the SAS log:

```
---+----1----+----2----+----3
phone=619-555-9340
phone=(213) 555-4820
phone=(49)12 34-56 78-90
```

**Example 4: 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 “$VARYINGw. Informat” on page 965 for more information.

**Example 5: 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 that is 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 6: 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 7: 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 8: 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

**Example 9: Working with Data in the Input Buffer**  
The _INFILE_ variable always contains the most recent record that is read from an INPUT statement. This example illustrates the use of the _INFILE_ variable to read an entire record that you want to parse without using the INPUT statement.

The example file contains phone bill information. The numeric data, minutes, and charge are enclosed in angle brackets (< >).

filename phonbill host-specific-filename;
data _null_
    file phonbill;
    input line $80.;
    put line;
Data lines;
City Number Minutes Charge
Jackson 415-555-2384 <25> <2.45>
Jefferson 813-555-2356 <15> <1.62>
Joliet 913-555-3223 <65> <10.32>
;
run;

The following code reads each record and parses the record to extract the minute and charge values.

data _null_
   infile phonbill firstobs=2;
   input;
      city = scan(_infile_, 1, ' ');
      char_min = scan(_infile_, 3, ' ');
      char_min = substr(char_min, 2, length(char_min)-2);
      minutes = input(char_min, BEST12.);
      put city= minutes=;
run;

The program writes the following lines to the SAS log:

----+----1----+----2----+----3
city=Jackson minutes=25
city=Jefferson minutes=15
city=Joliet minutes=65

The INPUT statement in the following code reads a record from the file. The automatic _INFILE_ variable is used in the PUT statement to write the record to the log.

data _null_
   infile phonbill;
   input;
      put _infile_;
run;

The program writes the following lines to the SAS log:

----+----1----+----2----+----3----+----4
City Number Minutes Charge
Jackson 415-555-2384 <25> <2.45>
Jefferson 813-555-2356 <15> <1.62>
Joliet 913-555-3223 <65> <10.32>

In the following code, the first INPUT statement reads and holds the record in the input buffer. The _INFILE_= option removes the angle brackets (< >) from the numeric data. The second INPUT statement parses the value in the buffer.

data _null_
   length city number $16. minutes charge 8;
   infile phonbill firstobs=2;
   input $;
   _infile_ = compress(_infile_, '<>);
   input city number minutes charge;
   put city= number= minutes= charge=;
run;

The program writes the following lines to the SAS log:
Example 10: Accessing the Input Buffers of Multiple Files  This example uses both the _INFILE_ automatic variable and the _INFILE_= option to read multiple files and access the input buffers for each of them. The following code creates four files: three data files and one file that contains the names of all the data files. The second DATA step reads the filenames file, opens each data file, and writes the contents to the log. Because the PUT statement needs _INFILE_ for the filenames file and the data file, one of the _INFILE_ variables is referenced with “fname.”

```sas
data _null_
   do i = 1 to 3;
      fname= 'external-data-file' || put(i,1.) || '.dat';
      file datfiles filevar=fname;
      do j = 1 to 5;
         put i j;
      end;
      file 'external-filenames-file';
      put fname;
   end;
run;

data _null_
   infile 'external-filenames-file' _infile_=fname;
   input;
   infile datfiles filevar=fname end=eof;
   do while(^eof);
      input;
      put fname _infile_;
   end;
run;
```

The program writes the following lines to the SAS log:

```
NOTE: The infile 'external-filenames-file' is:
   File Name=external-filenames-file,
   RECFM=V, LRECL=256

NOTE: The infile DATFILES is:
   File Name=external-data-file1.dat,
   RECFM=V, LRECL=256

external-data-file1.dat 1 1
external-data-file1.dat 1 2
external-data-file1.dat 1 3
external-data-file1.dat 1 4
external-data-file1.dat 1 5

NOTE: The infile DATFILES is
   File Name=external-data-file2.dat,
```
Example 11: Specifying an Encoding When Reading an External File
This example creates a SAS data set from an external file. The external file's encoding is in UTF-8, and the current SAS session encoding is WLatin1. By default, SAS assumes that the external file is in the same encoding as the session encoding, which causes the character data to be written to the new SAS data set incorrectly.

To tell SAS what encoding to use when reading the external file, specify the ENCODING= option. When you tell SAS that the external file is in UTF-8, SAS then transcodes the external file from UTF-8 to the current session encoding when writing to the new SAS data set. Therefore, the data is written to the new data set correctly in WLatin1.

libname myfiles 'SAS-data-library';
filename extfile 'external-file';
data myfiles.unicode;
infile extfile encoding="utf-8";
input Make $ Model $ Year;
run;

See Also

Statements:
"FILENAME Statement" on page 1172
"INPUT Statement" on page 1248
"PUT Statement" on page 1344

INFORMAT Statement

Associates informats with variables
Valid: in a DATA step or PROC step
Category: Information  
Type: Declarative

Syntax

INFORMAT variable-1 < . . . variable-n> <informat>;
INFORMAT <variable-1> < . . . variable-n> <DEFAULT=default-informat>;
INFORMAT variable-1 < . . . variable-n> informat <DEFAULT=default-informat>;

Arguments

variable
names one or more variables to associate with an informat. You must specify at least one variable when specifying an informat or when including no other arguments. Specifying a variable is optional when using a DEFAULT= informat specification.

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 1248.

informat
specifies the informat for reading the values of the variables that are listed in the INFORMAT statement.

Tip: If an informat is associated with a variable by using the INFORMAT statement, and that same informat is not associated with that same variable in the INPUT statement, then that informat will 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 Statement, List” on page 1269.

See Also: “Informats by Category” on page 941
Featured in: Example 2 on page 1247

DEFAULT= default-informat
specifies a temporary default informat for reading values of the variables that are listed in the INFORMAT statement. If no variable is specified, then the DEFAULT= informat specification applies a temporary default informat for reading values of all the variables of that type included in the DATA step. Numeric informats are applied to numeric variables, and character informats are applied to character variables. 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.

Default: If you omit DEFAULT=, SAS uses w.d as the default numeric informat and $w. as the default character informat.

Restriction: Use this argument only in a DATA step.

Tip: A DEFAULT= specification can occur anywhere in an INFORMAT statement. It can specify either a numeric default, a character default, or both.
**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 *Base SAS Procedures Guide* for more information.

**Examples**

**Example 1: Specifying Default Informats**  This example uses an INFORMAT statement to associate a default numeric informat:

```sas
data tstinfmt;
  informat default=3.1;
```
Statements △ INFORMAT Statement 1247

Statements

INFORMAT Statement 1247

input x;
put x;
datalines;
111
222
333;

The PUT statement produces these results:

11.1
22.2
33.3

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 7.7

The SAS System

CONTENTS PROCEDURE

-----Alphabetic List of Variables and Attributes-----

<table>
<thead>
<tr>
<th>#</th>
<th>Variable</th>
<th>Type</th>
<th>Len</th>
<th>Pos</th>
<th>Informat</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>FirstName</td>
<td>Char</td>
<td>15</td>
<td>16</td>
<td>$15.</td>
</tr>
<tr>
<td>2</td>
<td>LastName</td>
<td>Char</td>
<td>15</td>
<td>31</td>
<td>$15.</td>
</tr>
<tr>
<td>3</td>
<td>n1</td>
<td>Num</td>
<td>8</td>
<td>0</td>
<td>6.2</td>
</tr>
<tr>
<td>4</td>
<td>n2</td>
<td>Num</td>
<td>8</td>
<td>8</td>
<td>7.3</td>
</tr>
</tbody>
</table>

The SAS System

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.

```sas
data rtest;
  set rtest;
  informat x;
run;
```

See Also

Statements:

- “ATTRIB Statement” on page 1111
- “INPUT Statement” on page 1248
- “INPUT Statement, List” on page 1269

INPUT Statement

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)>@@;
```

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 @.

For an example, see Example 2 on page 1259.

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 Informs” on page 1267

$ 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 1259

**pointer-control**

moves the input pointer to a specified line or column in the input buffer.

**See:** “Column Pointer Controls” on page 1250 and “Line Pointer Controls” on page 1251

**column-specifications**

specifies the columns of the input record that contain the value to read.

**Tip:** Informats are ignored. Only standard character and numeric data can be read correctly with this method.

**See:** “Column Input” on page 1253

**Featured in:** Example 1 on page 1259

**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 1271

**See:** “Format Modifiers for Error Reporting” on page 1252

**Featured in:** Example 6 on page 1261

**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 1253 and “List Input” on page 1253

**Featured in:** Example 2 on page 1268

(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 Informs” on page 1267

@ 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 1255

**Featured in:** Example 3 on page 1259
@@ 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, or when a record needs to be reread on the next iteration of the DATA step.

See Also: “Using Line-Hold Specifiers” on page 1255
Featured in: Example 4 on page 1260

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:

\[
\text{input @15 name $10.};
\]

Featured in: Example 7 on page 1262

@\(\text{numeric-variable}\)
moves the pointer to the column given by the value of \(\text{numeric-variable}\).

Range: a positive integer

Tip: If \(\text{numeric-variable}\) is not an integer, SAS truncates the decimal value and only uses the integer value. If \(\text{numeric-variable}\) is zero or negative, the pointer moves to column 1.

Example: The value of the variable A moves the pointer to column 15:

\[
a=15;
\text{input @a name $10.};
\]

Featured in: Example 5 on page 1261

@\((\text{expression})\)
moves the pointer to the column that is given by the value of \(\text{expression}\).

Restriction: \(\text{Expression}\) must result in a positive integer.

Tip: If the value of \(\text{expression}\) 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:

\[
b=5;
\text{input @(b*3) name $10.};
\]

@\('character-string'\)
locates the specified series of characters in the input record and moves the pointer to the first column after \(\text{character-string}\).

@\(\text{character-variable}\)
locates the series of characters in the input record that is given by the value of \(\text{character-variable}\) 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 non-blank column after the value of WEEKDAY:

```sas
input @1 day 1. @5 weekday $10.
@1 @weekday sales 8.2;
```

Featured in: Example 6 on page 1261

@ 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 1261

\( +n \)

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:

```sas
input @23 length 4. +5 width 4.;
```

Featured in: Example 7 on page 1262

\( +\text{numeric-variable} \)

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 1262

\( +(\text{expression}) \)

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 1229.
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.

**See Also:** “How Invalid Data is Handled” on page 1258

`??` 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.

**See Also:** “How Invalid Data is Handled” on page 1258

### 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.
Operating Environment Information: LOG files that are generated under z/OS and captured with PROC PRINTTO contain an ASA control character in column 1. If you are using the INPUT statement to read a LOG file that was generated under z/OS, you must account for this character if you use column input or column pointer controls.

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 Statement, Column” on page 1263.

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 Statement, List” on page 1269.

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 income comma6.;
```

This INPUT statement reads these data records correctly:

```
----+----1----+----2----+
Peterson 21,000
Morgan 17,132
```
The pointer control of +2 moves the input pointer to the field that contains the value for the variable INCOME. For more information, see “INPUT Statement, Formatted” on page 1265.

**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:

```
name=Peterson age=21
name=Morgan age=17
```

For more information, see “INPUT Statement, Named” on page 1275.

**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:

```
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 7.6 on page 1255 lists the pointer controls that are available with the INPUT statement.
Table 7.6  Pointer Controls Available in the INPUT Statement

<table>
<thead>
<tr>
<th>Pointer Controls</th>
<th>Relative</th>
<th>Absolute</th>
</tr>
</thead>
<tbody>
<tr>
<td>column pointer controls</td>
<td>$+n$</td>
<td>$@n$</td>
</tr>
<tr>
<td></td>
<td>$+\text{numeric-variable}$</td>
<td>$@\text{numeric-variable}$</td>
</tr>
<tr>
<td></td>
<td>$(\text{expression})$</td>
<td>$(\text{expression})$</td>
</tr>
<tr>
<td></td>
<td>'@\text{character-string}'</td>
<td>'@\text{character-variable}'</td>
</tr>
<tr>
<td></td>
<td>'@(\text{character-expression})'</td>
<td></td>
</tr>
<tr>
<td>line pointer controls</td>
<td>$/$</td>
<td>$#n$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$#\text{numeric-variable}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$#(\text{expression})$</td>
</tr>
<tr>
<td>line-hold specifiers</td>
<td>$@$</td>
<td>(not applicable)</td>
</tr>
<tr>
<td></td>
<td>$@@$</td>
<td>(not applicable)</td>
</tr>
</tbody>
</table>

**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 within 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 “Reading More Than One Record per Observation” on page 1256.

**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 @)
- a record needs to be reread on the next iteration of the DATA step (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:

```plaintext
  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.;
  input region $7. @12 jansales 5.;
  ```

To read a value for the variable REGION, the 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:

\[
\text{input @31 age 3. #3 id 3-4 #2 @6 name $20. ;}
\]

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:

\[
\text{input name$ 1-10 #2 age 13-14 #4 ;}
\]

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:

\[
\text{input a / b #1 @52 c #2 ;}
\]

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 1229.

**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:

\[
\text{NOTE: SAS went to a new line when INPUT statement reached past the end of a line.}
\]

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:

\[
\text{data test;}
\text{input a @(a-3) b;}
\text{datalines;}
2
;
\]
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 is 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= System Option” on page 1542.
- 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 Statement” on page 1224.

**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 1261.

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:

```sas
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 ...
; 
```

<table>
<thead>
<tr>
<th>The values for ...</th>
<th>Are read with ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idno, Startwght, Endwght</td>
<td>list input</td>
</tr>
<tr>
<td>Name</td>
<td>formatted input</td>
</tr>
<tr>
<td>Team</td>
<td>column input</td>
</tr>
</tbody>
</table>

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:

```sas
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
```
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.

```sas
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.

```
The SAS System 1

<table>
<thead>
<tr>
<th>OBS</th>
<th>Course</th>
<th>Professor</th>
<th>Name</th>
<th>Id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>HIST101</td>
<td>Watson</td>
<td>Williams</td>
<td>459</td>
</tr>
<tr>
<td>2</td>
<td>HIST101</td>
<td>Watson</td>
<td>Flores</td>
<td>5423</td>
</tr>
<tr>
<td>3</td>
<td>MATH202</td>
<td>Sen</td>
<td>Lee</td>
<td>7085</td>
</tr>
</tbody>
</table>
```

**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.

```sas
data test;
  input name $ age @@;
datalines;
```
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

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>New York</td>
<td>1</td>
<td>USA</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>Cary</td>
<td>1</td>
<td>USA</td>
<td>2274</td>
</tr>
<tr>
<td>3</td>
<td>Chicago</td>
<td>1</td>
<td>USA</td>
<td>37</td>
</tr>
<tr>
<td>22</td>
<td>Tokyo</td>
<td>5</td>
<td>ASIA</td>
<td>80</td>
</tr>
<tr>
<td>5</td>
<td>Vancouver</td>
<td>2</td>
<td>CANADA</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>Milano</td>
<td>4</td>
<td>EUROPE</td>
<td>123</td>
</tr>
</tbody>
</table>

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 1261, 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');
```
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 non-blank 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 7.9   Data Set Office2

<table>
<thead>
<tr>
<th>OBS</th>
<th>City</th>
<th>Location</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>New York</td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>Cary</td>
<td>1</td>
<td>2274</td>
</tr>
<tr>
<td>3</td>
<td>Chicago</td>
<td>1</td>
<td>37</td>
</tr>
<tr>
<td>4</td>
<td>Vancouver</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>Milano</td>
<td>4</td>
<td>123</td>
</tr>
</tbody>
</table>

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 Statement” on page 1102
- “INPUT Statement, Column” on page 1263
- “INPUT Statement, Formatted” on page 1265
- “INPUT Statement, List” on page 1269
- “INPUT Statement, Named” on page 1275
INPUT Statement, 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 16:

   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:

<table>
<thead>
<tr>
<th>Input Data</th>
<th>Statement</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>2314</td>
<td>input number 1-5 .2;</td>
<td>23.14</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>.02</td>
</tr>
<tr>
<td>400</td>
<td></td>
<td>4.00</td>
</tr>
<tr>
<td>-140</td>
<td></td>
<td>-1.40</td>
</tr>
</tbody>
</table>
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:** “Pointer Controls” on page 1254.

@@ 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 1255.

**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 Statement, Formatted” on page 1265.

* See SAS Language Reference: Concepts for the definition of standard and nonstandard data values.
**Missing Values**  Missing data do not require a placeholder. 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 might 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 1235.

**Examples**

This DATA step demonstrates how to read input data records with column input:

```sas
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 Statement” on page 1248

---

**INPUT Statement, 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**

```sas
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 1250 and “Line Pointer Controls” on page 1251

`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 1268

`(variable-list)`

specifies a list of variables that are assigned input values.

See: “How to Group Variables and Informats” on page 1267

Featured in: Example 2 on page 1268

`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 929

Featured in: Example 1 on page 1268

`(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:

```plaintext
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 1267

Featured in: Example 2 on page 1268

`@`

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 1255

`@@`
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 1255

### 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 of Informats” on page 932 for descriptions of SAS informats.

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 Statement” on page 1248.

#### 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 might need to use the TRUNCOVER option in the INFILE statement. For more information, see “Reading Past the End of a Line” on page 1235.

#### 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:

```sas
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

```sas
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

---

* See *SAS Language Reference: Concepts* for information on standard and nonstandard data values.
statement ignores any directions that remain in the informat list. For an example, see Example 3 on page 1269.

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 1271.

**Examples**

**Example 1: Formatted Input with Pointer Controls** This INPUT statement uses informats and pointer controls:

```sas
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:

```sas
data scores;
  input (name score1-score5) ($10. 5*4.);
datalines;
Whittaker 121 114 137 156 142
Smythe 111 97 122 143 127
;```
Example 3: Including More Informat Specifications Than Necessary

This informat list includes more specifications than are necessary when the INPUT statement executes:

```sas
data test;
  input (x y z) (2.,+1);
  datalines;
  2  4  6
  0  3  0
;```

The INPUT statement reads the value of X with the 2. informat. Then, the +1 column pointer control moves the pointer forward one column. Next, 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.

See Also

Statements:

- “INPUT Statement” on page 1248
- “INPUT Statement, List” on page 1269

INPUT Statement, 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 1250 and “Line Pointer Controls” on page 1251

  Featured in:  Example 2 on page 1273

- `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 1273

& 
indicates that a character value may have one or more single embedded blanks. This format modifier reads the value from the next non-blank 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 1272

**Featured in:** Example 2 on page 1273

: 
enables you to specify an informat that the INPUT statement uses to read the variable value. For a character variable, this format modifier reads the value from the next non-blank 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. For a numeric variable, this format modifier reads the value from the next non-blank column until the pointer reaches the next blank column or the end of the data line, whichever comes first.

**Tip:** If the length of the variable has not been previously defined, then 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, then the value is truncated to the length of variable.

**See:** “Modified List Input” on page 1272

**Featured in:** Example 3 on page 1274 and Example 5 on page 1274

~ 
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 1272

**Featured in:** Example 5 on page 1274

**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 of Informats” on page 932

**Featured in:** Example 3 on page 1274 and Example 5 on page 1274

@ 
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 1255

@@ 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 1255

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 enable 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 1272.

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.*

* See SAS Language Reference: Concepts for the information about standard and nonstandard data values.
**Modified List Input** List input is more versatile when you use format modifiers. The format modifiers are as follows:

<table>
<thead>
<tr>
<th>Format Modifier</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>reads character values that contain embedded blanks.</td>
</tr>
<tr>
<td>:</td>
<td>reads data values that need the additional instructions that informats can provide but that are not aligned in columns. **</td>
</tr>
<tr>
<td>~</td>
<td>reads delimiters within quoted character values as characters and retains the quotation marks.</td>
</tr>
</tbody>
</table>

** Use formatted input and pointer controls to quickly read data values that are 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 1274. 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.

**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:

```plaintext
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:

```plaintext
input item $10. +1 amount comma5.;
```

To read this data correctly with formatted input, the second data value must occur after the 10th column of the first value, as shown here:

```
-----++++1-----++++2
trucks 1,382
```
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 Contains 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:

```plaintext
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:

```plaintext
input name $ score1 score2 score3;
```

**Example 2: Reading Character Data That Contains Embedded Blanks**  
The INPUT statement in this DATA step uses the & format modifier with list input to read character values that contain embedded blanks.

```plaintext
data list;
   infile file-specification;
   input name $ & score;
run;
```

It can read these input data records:

```
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:

```plaintext
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:

```sas
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 Comma-Delimited Data with List Input and an Informat

This DATA step uses the DELIMITER= option in the INFILE statement to read list input values that are separated by commas instead of blanks. The example uses an informat to read the date, and a format to write the date.

```sas
options pageno=1 nodate ls=80 ps=64;
data scores2;
  length Team $ 14;
  infile datalines delimiter=',';
  input Name $ Score1-Score3 Team $ Final_Date:MMDDYY10.;
  format final_date weekdate17.;
datalines;
Joe,11,32,76,Red Racers,2/3/2003
Mitchell,13,29,82,Blue Bunnies,4/5/2003
Susan,14,27,74,Green Gazelles,11/13/2003
;
proc print data=scores2;
  var Name Team Score1-Score3 Final_Date;
  title 'Soccer Player Scores';
run;
```

Output 7.10  Output from Comma-Delimited Data

<table>
<thead>
<tr>
<th>Obs</th>
<th>Name</th>
<th>Team</th>
<th>Score1</th>
<th>Score2</th>
<th>Score3</th>
<th>Final_Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Joe</td>
<td>Red Racers</td>
<td>11</td>
<td>32</td>
<td>76</td>
<td>Mon, Feb 3, 2003</td>
</tr>
<tr>
<td>2</td>
<td>Mitchell</td>
<td>Blue Bunnies</td>
<td>13</td>
<td>29</td>
<td>82</td>
<td>Sat, Apr 5, 2003</td>
</tr>
<tr>
<td>3</td>
<td>Susan</td>
<td>Green Gazelles</td>
<td>14</td>
<td>27</td>
<td>74</td>
<td>Thu, Nov 13, 2003</td>
</tr>
</tbody>
</table>

Example 5: Reading Delimited Data with Modified List Input

This DATA step uses the DSDF 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 string that is enclosed in quotation marks as a character instead of as a delimiter.

```sas
data scores;
  infile datalines dsd;
  input Name : $9. Score1-Score3
```
Team - $25. Div $;
datalines;
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 7.11 SCORES Data Set

<table>
<thead>
<tr>
<th>OBS</th>
<th>Name</th>
<th>Score1</th>
<th>Score2</th>
<th>Score3</th>
<th>Team</th>
<th>Div</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Joseph</td>
<td>11</td>
<td>32</td>
<td>76</td>
<td>&quot;Red Racers, Washington&quot;</td>
<td>AAA</td>
</tr>
<tr>
<td>2</td>
<td>Mitchel</td>
<td>13</td>
<td>29</td>
<td>82</td>
<td>&quot;Blue Bunnies, Richmond&quot;</td>
<td>AAA</td>
</tr>
<tr>
<td>3</td>
<td>Sue Ellen</td>
<td>14</td>
<td>27</td>
<td>74</td>
<td>&quot;Green Gazelles, Atlanta&quot;</td>
<td>AA</td>
</tr>
</tbody>
</table>

See Also

Statements:
- “INFILE Statement” on page 1224
- “INPUT Statement” on page 1248
- “INPUT Statement, Formatted” on page 1265

INPUT Statement, 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 1250 and “Line Pointer Controls” on page 1251

**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 1277

$ 

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 1277

**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 929

**Featured in:** Example 1 on page 1277

**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 1255

@@

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 1255

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 1277.

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,

```sas
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,

  ```sas
  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 1278.

- 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:

```sas
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
;```

```sas
Example 2: Using Named Input with Embedded Blanks**
```
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:

```sas
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 Statement” on page 1248

---

**KEEP Statement**

Includes variables in output SAS data sets

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

**Syntax**

```sas
KEEP variable-list;
```

**Arguments**

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.

```sas
data average;
  keep name avg;
  infile file-specification;
  input name $ score1-score20;
  avg=mean(of score1-score20);
run;
```

**See Also**

Data Set Option:
- “KEEP= Data Set Option” on page 30

Statements:
- “DROP Statement” on page 1152
- “IF Statement, Subsetting” on page 1214
- “RETAIN Statement” on page 1383
**LABEL Statement**

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 *Base 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";`
Examples:

Example 1: Adding Labels

```sas
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:

```sas
data rtest;
  set rtest;
  label x=' ';
run;
```

See Also

Statement:

“ATTRIB Statement” on page 1111

---

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.

```sas
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 Statement” on page 1213
“LINK Statement” on page 1296

Statement Options:
HEADER= option in the FILE statement on page 1161
EOF= option in the INFILE statement on page 1226

---

**LEAVE Statement**

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.

```plaintext
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 PT TECH
Harrell 1250 1975 PT HR
Richards 1002 1990 PT DEV
Kelly 85 1981 PT PUB
Stone 091 1990 PT MAIT
;
```

LENGTH Statement

Specifies the number of bytes for storing variables

Valid: in a DATA step
Category: Information
Type: Declarative
See: LENGTH Statement in the documentation for your operating environment.
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.

```sas
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 7.12

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Len</th>
<th>Pos</th>
<th>Informat</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 FirstName</td>
<td>Char</td>
<td>15</td>
<td>8</td>
<td>$15.</td>
</tr>
<tr>
<td>2 LastName</td>
<td>Char</td>
<td>15</td>
<td>23</td>
<td>$15.</td>
</tr>
<tr>
<td>3 n1</td>
<td>Num</td>
<td>4</td>
<td>0</td>
<td>6.2</td>
</tr>
<tr>
<td>4 n2</td>
<td>Num</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5 name</td>
<td>Char</td>
<td>25</td>
<td>38</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OBS</th>
<th>FirstName</th>
<th>LastName</th>
<th>n1</th>
<th>n2</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alexander</td>
<td>Robinson</td>
<td>0.35000</td>
<td>11</td>
<td>Robinson, Alexander</td>
</tr>
</tbody>
</table>
LIBNAME Statement

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: LIBNAME Statement in the documentation for your operating environment

Syntax

1. LIBNAME libref <engine> 'SAS-data-library' < options > <engine/host-options>;
2. LIBNAME libref CLEAR | _ALL_ CLEAR;
3. LIBNAME libref LIST | _ALL_ LIST;
4. LIBNAME libref <engine> (library-specification-1 < . . . library-specification-n>) < options >;

Arguments

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 you are 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 about specifying the physical names of files, see the SAS documentation for your operating environment.
library-specification

is two or more SAS data libraries that are 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'

is 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 1292

See Also: “Rules for Library Concatenation” on page 1291

engine

is an engine name.

Tip: Usually, 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, then 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 is 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.
COMPRESS=NO | YES | CHAR | BINARY controls the compression of observations in output SAS data sets for a SAS data library.

NO specifies that the observations in a newly created SAS data set be uncompressed (fixed-length records).

YES | CHAR specifies that the observations in a newly created SAS data set be compressed (variable-length records) by SAS using RLE (Run Length Encoding). RLE compresses observations by reducing repeated consecutive characters (including blanks) to two-byte or three-byte representations.

Tip: Use this compression algorithm for character data.

BINARY specifies that the observations in a newly created SAS data set be compressed (variable-length records) by SAS using RDC (Ross Data Compression). RDC combines run-length encoding and sliding-window compression to compress the file.

Tip: This method is highly effective for compressing medium to large (several hundred bytes or larger) blocks of binary data (numeric variables). Because the compression function operates on a single record at a time, the record length needs to be several hundred bytes or larger for effective compression.

CVPBYTES=bytes specifies the number of bytes to expand character variable lengths when processing a SAS data file that requires transcoding.


CVPENGINE|CVPENG=engine specifies the engine to use in order to process the SAS file.


CVPMULTIPLIER|CVPMULT=multiplier specifies a multiplier value in order to expand character variable lengths when processing a SAS data file that requires transcoding.


INENCODING=ANY | ASCIANY | EBCDICANY | encoding-value overrides the encoding when you are reading (input processing) SAS data sets in the SAS data library.


OUTENCODING= OUTENCODING=ANY | ASCIANY | EBCDICANY | encoding-value overrides the encoding when you are creating (output processing) SAS data sets in the SAS data library.


OUTREP=format
specifies the default data representation for the SAS data library. New data sets that are stored in this SAS data library are written in the specified data representation. Data representation is the format in which data is represented in a computer architecture or in an operating environment. For example, on an IBM PC, character data is represented by its ASCII encoding and byte-swapped integers. Native data representation refers to an environment in which the data representation is comparable to the CPU that is accessing the file. For example, a file that is in Windows data representation is native to the Windows operating environment.

Specifying this option enables you to create files within the native environment by using a foreign environment data representation. For example, in a UNIX environment, you can create a SAS data set in Windows data representation. Existing data sets that are written to the library are given the new format.

Values for OUTREP= are listed in the following table:

<table>
<thead>
<tr>
<th>OUTREP= Value</th>
<th>Alias</th>
<th>Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALPHA_TRU64</td>
<td>ALPHA_OSF</td>
<td>Compaq Tru64 UNIX</td>
</tr>
<tr>
<td>ALPHA_VMS_32</td>
<td>ALPHA_VMS</td>
<td>OpenVMS Alpha on 32-bit platform</td>
</tr>
<tr>
<td>ALPHA_VMS_64</td>
<td></td>
<td>OpenVMS Alpha on 64-bit platform</td>
</tr>
<tr>
<td>HP_IA64</td>
<td>HP_ITANIUM</td>
<td>HP UX on Itanium 64-bit platform</td>
</tr>
<tr>
<td>HP_UX_32</td>
<td>HP_UX</td>
<td>HP UX on 32-bit platform</td>
</tr>
<tr>
<td>HP_UX_64</td>
<td></td>
<td>HP UX on 64-bit platform</td>
</tr>
<tr>
<td>INTEL_ABI</td>
<td></td>
<td>ABI UNIX on Intel 32-bit platform</td>
</tr>
<tr>
<td>LINUX_32</td>
<td>LINUX</td>
<td>Linux for Intel Architecture on 32-bit platform</td>
</tr>
<tr>
<td>MIPS_ABI</td>
<td></td>
<td>ABI UNIX on 32-bit platform</td>
</tr>
<tr>
<td>MVS_32</td>
<td>MVS</td>
<td>z/OS on 32-bit platform</td>
</tr>
<tr>
<td>OS2</td>
<td></td>
<td>OS/2 on Intel 32-bit platform</td>
</tr>
<tr>
<td>RS_6000_AIX_32</td>
<td>RS_6000_AIX</td>
<td>AIX UNIX on 32-bit RS/6000</td>
</tr>
<tr>
<td>RS_6000_AIX_64</td>
<td></td>
<td>AIX UNIX on 64-bit RS/6000</td>
</tr>
<tr>
<td>SOLARIS_32</td>
<td>SOLARIS</td>
<td>Sun Solaris on 32-bit platform</td>
</tr>
<tr>
<td>SOLARIS_64</td>
<td></td>
<td>Sun Solaris on 64-bit platform</td>
</tr>
<tr>
<td>VAX_VMS</td>
<td></td>
<td>VAX VMS</td>
</tr>
<tr>
<td>WINDOWS_32</td>
<td>WINDOWS</td>
<td>Microsoft Windows on 32-bit platform</td>
</tr>
<tr>
<td>WINDOWS_64</td>
<td></td>
<td>Microsoft Windows 64-bit Edition</td>
</tr>
</tbody>
</table>

* It is recommended that you use the current values. The aliases are available for compatibility only.

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 replace 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 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:

```sas
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 mistake, 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= Data Set Option” on page 47

---

**Engine Host Options**

`engine-host-options` are one or more options that are 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**

**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 the libref 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:

```sas
LIBNAME libref 'SAS-data-library';
```

See Example 1 on page 1292.

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, then SAS uses that engine as the default. In either situation, you can override the default by specifying another engine with the ENGINE= system option:

```sas
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.


**Disassociating a Libref from a SAS Data Library**  
To disassociate a libref from a SAS data library, use a LIBNAME statement by specifying the libref and the CLEAR option. You can clear a single, specified libref or all current librefs.

```
LIBNAME libref CLEAR | _ALL_ CLEAR;
```

**Writing 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;
```

**Concatenating 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 filename 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 filenames, or a combination of librefs and physical filenames. See Example 2 on page 1292.

**Implicitly 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 1293.

```
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 (non-concatenated) 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. Anytime a list of SAS files is displayed, only one occurrence of a filename 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.
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.”

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.

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 enables 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 filename. 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.

```sas
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:

  ```sas
  libname allmine ('file-1' 'file-2' 'file-3');
  ```

- This example assigns librefs to two SAS data libraries, one that contains SAS 6 files and one that contains SAS 9 files. This technique is useful for updating your files and applications from SAS 6 to SAS 9, while allowing you to have convenient access to both sets of files:

  ```sas
  libname v6 'v6--data-library';
  libname v9 'v9--data-library';
  libname allmine (v9 v6);
  ```
This example shows that you can specify both librefs and physical filenames in the same concatenation specification:

```sas
libname allmine (v9 v6 'some-filename');
```

**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:

```sas
libname allmine ('file-1' 'file-2' 'file-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 Statement” on page 1120.

**Example 4: Permanently Storing Data Sets with One-Level Names**

If you want the convenience of specifying only a one-level name for permanent, not temporary, SAS files, then use the USER= system option. This example stores the data set QUARTER1 permanently without using a LIBNAME statement first to assign a libref to a storage location:

```sas
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**

Data Set Options:  
“ENCODING= Data Set Option” on page 18

Statements:  
“CATNAME Statement” on page 1120 for a discussion of explicitly concatenating SAS catalogs  
“FILENAME Statement” on page 1172  
“LIBNAME Statement” for Scalable Performance Data (SPD) in SAS Scalable Performance Data Engine: Reference  
LIBNAME Statement, 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

See: LIBNAME Statement for Relational Databases in SAS/ACCESS for Relational Databases: Reference

LIBNAME Statement for SAS/CONNECT Remote Library Services

Associates a libref with a SAS data library that is located on the server for client access

Valid: Client Session

Category: Data Access


LIBNAME Statement for the Metadata Engine

Associates a SAS libref with the metadata that is in a SAS Metadata Repository on the SAS Metadata Server

Valid: Anywhere

Category: Data Access

LIBNAME Statement, SASEDOC

Associates a SAS libref with one or more ODS output objects that are stored in an ODS document

Valid: Anywhere
Category: ODS: Output Control

LIBNAME Statement for SAS/CONNECT TCP/IP Pipe

Associates a libref with a TCP/IP pipe (instead of a physical disk device) for processing input and output. The SASESOCK engine is required for SAS/CONNECT applications that implement MP CONNECT with piping.

Valid: Client Session and Server Session
Category: Data Access

LIBNAME Statement for SAS/SHARE

In a client session, associates a libref with a SAS data library that is located on the server for client access. In a server session, pre-defines a server library that clients are permitted to access.

Valid: Client and Server Sessions
Category: Data Access

LIBNAME Statement for Scalable Performance Data

Associates a SAS libref with a SAS data library for rapid processing of very large data sets by multiple CPUs

Valid: Anywhere
Category: Data Access
See: LIBNAME Statement for the SPD engine in SAS Scalable Performance Data Engine: Reference
LIBNAME Statement for the XML Engine

Associates a SAS libref with the physical location of an XML document

Valid: Anywhere
Category: Data Access

LINK Statement

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.

```sas
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 Statement” on page 1145
“GO TO Statement” on page 1213
“Labels, Statement” on page 1281
“RETURN Statement” on page 1387

**LIST Statement**

Writes to the SAS log the input data record 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 record for the observation being processed to be written to the SAS log.

Details
The LIST statement operates only on data that is read with an INPUT statement; it has no effect on data that is 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.

For variable-length records (RECFM=V), SAS writes the record length at the end of the input line. SAS does not write the length for fixed-length records (RECFM=F), unless the amount of data read does not equal the record length (LRECL).

Comparisons

<table>
<thead>
<tr>
<th>Action</th>
<th>LIST Statement</th>
<th>PUT Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Writes when</td>
<td>at the end of each iteration of the DATA step</td>
<td>immediately</td>
</tr>
<tr>
<td>Writes what</td>
<td>the input data records exactly as they appear</td>
<td>the variables or literals specified</td>
</tr>
<tr>
<td>Writes where</td>
<td>only to the SAS log</td>
<td>to the SAS log, the SAS output destination, or to any external file</td>
</tr>
<tr>
<td>Works with</td>
<td>INPUT statement only</td>
<td>any data-reading statement</td>
</tr>
<tr>
<td>Handles hex values</td>
<td>automatically prints a hexadecimal value if it encounters an unprintable character</td>
<td>represents characters in hexadecimal only when a hex format is given</td>
</tr>
</tbody>
</table>

Examples

Example 1: Listing Records That Contain Missing Data

This example uses the LIST statement to write to the SAS log any input records that contain missing data. 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.

data employee;
    input ssn 1-9 #3 w2amt 1-6;
if w2amt=. then list;
datalines;
23456789
JAMES SMITH
356.79
345671234
Jeffrey Thomas
.
;

Output 7.13  Log Listing of Missing Data

The numbers 9, 10, and 11 are line numbers in the SAS log.

Example 2: Listing the Record Length of Variable-Length Records

This example uses as input an external file that contains variable-length ID numbers. The RECFM=V option is specified in the INFILE statement, and the LIST statement writes the records to the SAS log. When the file has variable-length records, as indicated by the RECFM=V option in this example, SAS writes the record length at the end of each record that is listed in the SAS log.

data employee;
  infile 'your-external-file' recfm=v;
  input id $;
  list;
run;

Output 7.14  Log Listing of Variable-Length Records and Record Lengths

See Also

Statement:

“PUT Statement” on page 1344
%LIST Statement

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 Statement” on page 1218
System Option:
“SPOOL System Option” on page 1606

LOCK Statement

Acquires and releases an exclusive lock on an existing SAS file

Valid:  Anywhere
Category:  Program Control
Restriction:  You cannot lock a SAS file that another SAS session is currently accessing (either from an exclusive lock or because the file is open).
Restriction:  The LOCK statement syntax is the same whether you issue the statement in a single-user environment or in a client/server environment. However, some LOCK statement functionality applies only to a client/server environment.

Syntax

LOCKS <libref><.member-name><.member-type | .entry-name.entry-type>> <LIST | QUERY | SHOW | CLEAR> ;

Arguments

libref
is a name that is associated with a SAS data library. The libref (library reference) must be a valid SAS name. If the libref is SASUSER or WORK, you must specify it.
Tip: In a single-user environment, you typically would not issue the LOCK statement to exclusively lock a library. To lock a library that is accessed via a multiuser SAS/SHARE server, see the LOCK statement in the SAS/SHARE User's Guide.

member-name
is a valid SAS name that specifies a member of the SAS data library that is associated with the libref.
Restriction: The SAS file must be created before you can request a lock. For information about locking a member of a SAS data library when the member does not exist, see the SAS/SHARE User's Guide.

member-type
is the type of SAS file to be locked. For example, valid values are DATA, VIEW, CATALOG, MDDB, and so on. The default is DATA.

entry-name
is the name of the catalog entry to be locked.
Tip: In a single-user environment, if you issue the LOCK statement to lock an individual catalog entry, the entire catalog is locked; you typically would not issue the LOCK statement to exclusively lock a catalog entry. To lock a catalog entry in a library that is accessed via a multiuser SAS/SHARE server, see the LOCK statement in the SAS/SHARE User's Guide.

**entry-type**

is the type of the catalog entry to be locked.

Tip: In a single-user environment, if you issue the LOCK statement to lock an individual catalog entry, the entire catalog is locked; you typically would not issue the LOCK statement to exclusively lock a catalog entry. To lock a catalog entry in a library that is accessed via a multiuser SAS/SHARE server, see the LOCK statement in the SAS/SHARE User's Guide.

**LIST | QUERY | SHOW**

writes to the SAS log whether you have an exclusive lock on the specified SAS file.

Tip: This option provides more information in a client/server environment. To use this option in a client/server environment, see the LOCK statement in the SAS/SHARE User's Guide.

**CLEAR**

releases a lock on the specified SAS file that was acquired by using the LOCK statement in your SAS session.

**Details**

**General Information** The LOCK statement enables you to acquire and release an exclusive lock on an existing SAS file. Once an exclusive lock is obtained, no other SAS session can read or write to the file until the lock is released. You release an exclusive lock by using the CLEAR option.

**Acquiring Exclusive Access to a SAS File in a Single-User Environment** Each time you issue a SAS statement or a procedure to process a SAS file, the file is opened for input, update, or output processing. At the end of the step, the file is closed. In a program with multiple tasks, a file could be opened and closed multiple times. Because multiple SAS sessions in a single-user environment can access the same SAS file, issuing the LOCK statement to acquire an exclusive lock on the file protects data while it is being updated in a multistep program.

For example, consider a nightly update process that consists of a DATA step to remove observations that are no longer useful, a SORT procedure to sort the file, and a DATASETS procedure to rebuild the file's indexes. If another SAS session accesses the file between any of the steps, the SORT and DATASETS procedures would fail, because they require member-level locking (exclusive) access to the file.

Including the LOCK statement before the DATA step provides the needed protection by acquiring exclusive access to the file. If the LOCK statement is successful, a SAS session that attempts to access the file between steps will be denied access, and the nightly update process runs uninterrupted. See Example 1 on page 1303.

**Return Codes for the LOCK Statement** The SAS macro variable SYSLCKRC contains the return code from the LOCK statement. The following actions result in a nonzero value in SYSLCKRC:

- You try to lock a file but cannot obtain the lock (for example, the file was in use or is locked by another SAS session).
- You use a LOCK statement with the LIST option to list a lock you do not have.
You use a LOCK statement with the CLEAR option to release a lock you do not have.

For more information about the SYSLCKRC SAS macro variable, see SAS Macro Language: Reference.

Comparisons

With SAS/SHARE software, you can also use the LOCK statement. Some LOCK statement functionality applies only to a client/server environment.

The CNTLLEV= data set option specifies the level at which shared update access to a SAS data set is denied.

Examples

Example 1: Locking a SAS File  The following SAS program illustrates the process of locking a SAS data set. Including the LOCK statement provides protection for the multistep program by acquiring exclusive access to the file. Any SAS session that attempts to access the file between steps will be denied access, which ensures that the program runs uninterrupted.

libname mydata 'SAS-data-library';

lock mydata.census; 1

data mydata.census; 2
   modify mydata.census;
      (statements to remove obsolete observations)
run;

proc sort force data=mydata.census; 3
   by CrimeRate;
run;

proc datasets library=mydata; 4
   modify census;
       index create CrimeRate;
quit;

lock mydata.census clear; 5

1 Acquires exclusive access to the SAS data set MYDATA.CENSUS.
2 Opens MYDATA.CENSUS to remove observations that are no longer useful. At the end of the DATA step, the file is closed. However, because of the exclusive lock, any other SAS session that attempts to access the file is denied access.
3 Opens MYDATA.CENSUS to sort the file. At the end of the procedure, the file is closed but not available to another SAS session.
4 Opens MYDATA.CENSUS to rebuild the file's index. At the end of the procedure, the file is closed but still not available to another SAS session.
5 Releases the exclusive lock on MYDATA.CENSUS. The data set is now available to other SAS sessions.
See Also

Data Set Option:
“CNTLLEV= Data Set Option” on page 13
For information on locking a data object in a library that is accessed via a multiuser SAS/SHARE server, see the LOCK statement in the SAS/SHARE User’s Guide.

LOSTCARD Statement

Resynchronizes the input data when SAS encounters a missing or invalid record in data that has 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.
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.

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:

```sas
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 6432
  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 7.15

```
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 Statement” on page 1216

---

**MERGE Statement**

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 “Definition of Data Set Options” on page 6.

**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, Combining, and Modifying 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, Combining, and Modifying 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:

```plaintext
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:

```plaintext
data inventory;
  merge stock orders;
  by partnum;
run;
```

See Also

Statements:
- “BY Statement” on page 1114
- “MODIFY Statement” on page 1310
- “SET Statement” on page 1399
- “UPDATE Statement” on page 1416
- “Reading, Combining, and Modifying SAS Data Sets” in SAS Language Reference: Concepts

MISSING Statement

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:

```plaintext
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 Statement” on page 1416

System Option:

“MISSING= System Option” on page 1565
MODIFY Statement

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

1. `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;

2. `MODIFY master-data-set <(data-set-option(s))> KEY=index </ UNIQUE>
   <NOBS=variable> <END=variable> ;

3. `MODIFY master-data-set <(data-set-option(s))> <NOBS=variable> POINT=variable;
4. `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: The following restrictions apply:

- 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.

**by-variable**

specifies one or more variables by which you identify corresponding observations.

**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) or the last observation of the transaction data set (for matching access). It is also set to 1 when MODIFY cannot find a match for a KEY= value (random access). 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=**

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 1314.

**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 that is listed on the KEY=option for all duplicates in the master data set.

If duplicates exist in the transaction data set, and they are consecutive, 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 1312

**Featured in:** Example 4 on page 1321, Example 5 on page 1322, and Example 6 on page 1324

**NOBS=**

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 1320

**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:

- a BY statement
- a 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 1320

**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 1322

**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

Matching 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 1318 shows the matching access method.

Duplicate 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.

Direct 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 1321 shows the direct-access-by-indexed-values method.

Duplicate 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 1322 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.

Direct (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 1320 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.

Sequential 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: The IORCMMSG function returns a formatted error message associated with
the current value of _IORC_. △

Example 6 on page 1324 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.

Remember 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. You can code multiple OUTPUT, REPLACE, and REMOVE statements to
  apply to one observation. However, once an OUTPUT, REPLACE, or REMOVE
  statement executes, the MODIFY statement must execute again before the next
  REPLACE or REMOVE statement executes.
You can use OUTPUT and REPLACE in the following example of conditional logic because only one of the REPLACE or OUTPUT statements executes per observation:

```sas
data master;
    modify master trans; by key;
    if _iorc_=0 then replace;
    else
        output;
run;
```

But you should not use multiple REPLACE operations on the same observation as in this example:

```sas
data master;
    modify master;
    x=1;
    replace;
    replace;
run;
```

You can code multiple OUTPUT statements per observation. However, be careful when you use multiple OUTPUT statements. It is possible to go into an infinite loop with just one OUTPUT statement.

```sas
data master;
    modify master;
    output;
run;
```

- 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 1325 shows how to use the OUTPUT, REMOVE, and REPLACE statements to write observations.

**Using MODIFY with Data Set Options**  If you use data set options (such as KEEP=) in your program, then use the options in the MODIFY statement for the master data set. Using data set options in the DATA statement might produce unexpected results.

**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 “Duplicate BY Values” on page 1313.
- 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 INVTRY.STOCK data set. INVTRY.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, INVTRY.STOCK contains a simple index on PARTNO. This DATA step creates INVTRY.STOCK:

```sas
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.:
dataines;
K89R seal  34 27ju195 245.00
```
Examples

Example 1: Modifying All Observations  This example replaces the date on all of the records in the data set INVTY.STOCK with the current date. It also replaces the value of the variable RECDATE with the current date for all observations in INVTY.STOCK:

```sas
data invty.stock;
  modify invty.stock;
  recdate=today();
run;

proc print data=invty.stock noobs;
  title 'INVTY.STOCK';
run;
```

Output 7.16  Results of Updating the RECDATE Field

```
INVTY.STOCK
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 09sep96 11.55
JD03  switch 383 09jan97 13.99
BV1E  timer  26 03jan97 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:

```sas
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.

```sas
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;

proc print data=invty.stock noobs;
  title 'INVTY.STOCK';
run;
```
Output 7.17  Results of Updating the INSTOCK and RECDATE Fields

<table>
<thead>
<tr>
<th>INVTY.STOCK</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARTNO DESC</td>
<td>INSTOCK</td>
</tr>
<tr>
<td>K89R seal</td>
<td>89</td>
</tr>
<tr>
<td>M4J7 sander</td>
<td>119</td>
</tr>
<tr>
<td>LK43 filter</td>
<td>164</td>
</tr>
<tr>
<td>MN21 brace</td>
<td>116</td>
</tr>
<tr>
<td>BC85 clamp</td>
<td>137</td>
</tr>
<tr>
<td>NCF3 valve</td>
<td>288</td>
</tr>
<tr>
<td>KJ66 cutter</td>
<td>8</td>
</tr>
<tr>
<td>UYN7 rod</td>
<td>319</td>
</tr>
<tr>
<td>JD03 switch</td>
<td>438</td>
</tr>
<tr>
<td>BV1E timer</td>
<td>53</td>
</tr>
</tbody>
</table>

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;
1 251.00
2 49.33
3 12.32
4 30.00
5 15.00
6 25.75
7 22.00
8 14.00
9 14.32
10 35.00
;

This DATA step updates INVTY.STOCK:

libname invty ‘SAS-data-library’;

data invty.stock;
  set newp;
  modify invty.stock point=tool_obs
    nob= 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;

proc print data=invty.stock noobs;
   title 'INVTY.STOCK';
run;

Output 7.18  Results of Updating the RECDATE and PRICE Fields

<table>
<thead>
<tr>
<th>PARTNO</th>
<th>DESC</th>
<th>INSTOCK</th>
<th>RECDATE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>K89R</td>
<td>seal</td>
<td>34</td>
<td>14MAR97</td>
<td>251.00</td>
</tr>
<tr>
<td>M4J7</td>
<td>sander</td>
<td>98</td>
<td>14MAR97</td>
<td>49.33</td>
</tr>
<tr>
<td>LK43</td>
<td>filter</td>
<td>121</td>
<td>14MAR97</td>
<td>12.32</td>
</tr>
<tr>
<td>MN21</td>
<td>brace</td>
<td>43</td>
<td>14MAR97</td>
<td>30.00</td>
</tr>
<tr>
<td>BC85</td>
<td>clamp</td>
<td>80</td>
<td>14MAR97</td>
<td>15.00</td>
</tr>
<tr>
<td>NCF3</td>
<td>valve</td>
<td>198</td>
<td>14MAR97</td>
<td>25.75</td>
</tr>
<tr>
<td>KJ66</td>
<td>cutter</td>
<td>6</td>
<td>14MAR97</td>
<td>22.00</td>
</tr>
<tr>
<td>UYN7</td>
<td>rod</td>
<td>211</td>
<td>14MAR97</td>
<td>14.00</td>
</tr>
<tr>
<td>JD03</td>
<td>switch</td>
<td>383</td>
<td>14MAR97</td>
<td>14.32</td>
</tr>
<tr>
<td>BV1E</td>
<td>timer</td>
<td>26</td>
<td>14MAR97</td>
<td>35.00</td>
</tr>
</tbody>
</table>

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 1318.

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;
proc print data=invty.stock noobs;
   title 'INVTY.STOCK';
run;

Output 7.19  Results of Updating the INSTOCK and RECDATE Fields by Using an Index

<table>
<thead>
<tr>
<th>PARTNO</th>
<th>DESC</th>
<th>INSTOCK</th>
<th>RECDATE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>K89R</td>
<td>seal</td>
<td>89</td>
<td>14MAR97</td>
<td>245.00</td>
</tr>
<tr>
<td>M4J7</td>
<td>sander</td>
<td>119</td>
<td>14MAR97</td>
<td>45.88</td>
</tr>
<tr>
<td>LK43</td>
<td>filter</td>
<td>164</td>
<td>14MAR97</td>
<td>10.99</td>
</tr>
<tr>
<td>MN21</td>
<td>brace</td>
<td>116</td>
<td>14MAR97</td>
<td>27.87</td>
</tr>
<tr>
<td>BC85</td>
<td>clamp</td>
<td>137</td>
<td>14MAR97</td>
<td>9.55</td>
</tr>
<tr>
<td>NCF3</td>
<td>valve</td>
<td>288</td>
<td>14MAR97</td>
<td>24.50</td>
</tr>
<tr>
<td>KJ66</td>
<td>cutter</td>
<td>8</td>
<td>14MAR97</td>
<td>19.77</td>
</tr>
<tr>
<td>UYN7</td>
<td>rod</td>
<td>319</td>
<td>14MAR97</td>
<td>11.55</td>
</tr>
<tr>
<td>JD03</td>
<td>switch</td>
<td>438</td>
<td>14MAR97</td>
<td>13.99</td>
</tr>
<tr>
<td>BV1E</td>
<td>timer</td>
<td>53</td>
<td>14MAR97</td>
<td>34.50</td>
</tr>
</tbody>
</table>

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;

proc print data=invty.stock noobs;
  title 'Results of Using the UNIQUE Option';
run;
Chapter 7

Output 7.20  Results of Updating the INSTOCK and RECDATE Fields by Using the UNIQUE Option

<table>
<thead>
<tr>
<th>PARTNO</th>
<th>DESC</th>
<th>INSTOCK</th>
<th>RECDATE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>K89R</td>
<td>seal</td>
<td>89</td>
<td>14MAR97</td>
<td>245.00</td>
</tr>
<tr>
<td>M4J7</td>
<td>sander</td>
<td>145</td>
<td>14MAR97</td>
<td>45.88</td>
</tr>
<tr>
<td>LK43</td>
<td>filter</td>
<td>164</td>
<td>14MAR97</td>
<td>10.99</td>
</tr>
<tr>
<td>MN21</td>
<td>brace</td>
<td>116</td>
<td>14MAR97</td>
<td>27.87</td>
</tr>
<tr>
<td>BC85</td>
<td>clamp</td>
<td>137</td>
<td>14MAR97</td>
<td>9.55</td>
</tr>
<tr>
<td>NCF3</td>
<td>valve</td>
<td>288</td>
<td>14MAR97</td>
<td>24.50</td>
</tr>
<tr>
<td>KJ66</td>
<td>cutter</td>
<td>8</td>
<td>14MAR97</td>
<td>19.77</td>
</tr>
<tr>
<td>UYN7</td>
<td>rod</td>
<td>319</td>
<td>14MAR97</td>
<td>11.55</td>
</tr>
<tr>
<td>JD03</td>
<td>switch</td>
<td>438</td>
<td>14MAR97</td>
<td>13.99</td>
</tr>
<tr>
<td>BV1E</td>
<td>timer</td>
<td>53</td>
<td>14MAR97</td>
<td>34.50</td>
</tr>
</tbody>
</table>

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:

```sas
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;
proc print data=invty.stock noobs;
title 'INVTY.STOCK Data Set';
run;

Output 7.21  The Updated INVTY.STOCK Data Set

<table>
<thead>
<tr>
<th>PARTNO</th>
<th>DESC</th>
<th>INSTOCK</th>
<th>RECDATE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>K89R</td>
<td>seal</td>
<td>48</td>
<td>14NOV96</td>
<td>245.00</td>
</tr>
<tr>
<td>M4J7</td>
<td>sander</td>
<td>122</td>
<td>23AUG96</td>
<td>47.98</td>
</tr>
<tr>
<td>LK43</td>
<td>filter</td>
<td>132</td>
<td>29JAN97</td>
<td>14.99</td>
</tr>
<tr>
<td>MN21</td>
<td>brace</td>
<td>52</td>
<td>09JAN97</td>
<td>27.87</td>
</tr>
<tr>
<td>BC85</td>
<td>clamp</td>
<td>92</td>
<td>09DEC96</td>
<td>10.00</td>
</tr>
<tr>
<td>NCF3</td>
<td>valve</td>
<td>198</td>
<td>20MAR96</td>
<td>24.50</td>
</tr>
<tr>
<td>KJ66</td>
<td>cutter</td>
<td>6</td>
<td>18JUN96</td>
<td>19.77</td>
</tr>
<tr>
<td>UYN7</td>
<td>rod</td>
<td>211</td>
<td>09SEP96</td>
<td>11.55</td>
</tr>
<tr>
<td>JD03</td>
<td>switch</td>
<td>383</td>
<td>09JAN97</td>
<td>13.99</td>
</tr>
<tr>
<td>BV1E</td>
<td>timer</td>
<td>26</td>
<td>03JAN97</td>
<td>34.50</td>
</tr>
<tr>
<td>ME34</td>
<td>cutter</td>
<td>8</td>
<td>14NOV96</td>
<td>14.50</td>
</tr>
</tbody>
</table>

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.

```sas
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;
proc print data=invty.stock noobs;
title 'New Prices for Stock Received in 1996';
run;
```

**Output 7.22** Output from Writing Observations to a Specific SAS Data Set

<table>
<thead>
<tr>
<th>PARTNO</th>
<th>DESC</th>
<th>INSTOCK</th>
<th>RECDATE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>LK43</td>
<td>filter</td>
<td>121</td>
<td>19MAY96</td>
<td>12.089</td>
</tr>
<tr>
<td>MN21</td>
<td>brace</td>
<td>43</td>
<td>10AUG96</td>
<td>30.657</td>
</tr>
<tr>
<td>BC85</td>
<td>clamp</td>
<td>80</td>
<td>16AUG96</td>
<td>10.505</td>
</tr>
<tr>
<td>NCF3</td>
<td>valve</td>
<td>198</td>
<td>20MAR96</td>
<td>26.950</td>
</tr>
<tr>
<td>KJ66</td>
<td>cutter</td>
<td>6</td>
<td>18JUN96</td>
<td>21.747</td>
</tr>
<tr>
<td>UYN7</td>
<td>rod</td>
<td>211</td>
<td>09SEP96</td>
<td>12.705</td>
</tr>
</tbody>
</table>

**See Also**

Statements:

- “OUTPUT Statement” on page 1341
- “REMOVE Statement” on page 1377
- “REPLACE Statement” on page 1381
- “UPDATE Statement” on page 1416

“Reading, Combining, and Modifying SAS Data Sets” in SAS Language Reference: Concepts

“The SQL Procedure” in the Base SAS Procedures Guide
_NEW_ Statement

Creates an instance of a DATA step component object

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

Syntax

variable = _NEW_ object(<argument_tag-1: value-1<, ...argument_tag-n: value-n>>);

Arguments

variable
specifies the variable name for the component object.

object
specifies the component object. It can be one of the following:

hash
indicates a hash object. The hash object provides a mechanism for quick data storage and retrieval. The hash object stores and retrieves data based on look-up keys. For more information about the hash object, see “Using the Hash Object” in SAS Language Reference: Concepts.

hiter
indicates a hash iterator object. The hash iterator object enables you to retrieve the hash object’s data in forward or reverse key order. For more information about the hash iterator object, see “Using the Hash Iterator Object” in SAS Language Reference: Concepts.

argument-tag
specifies the information that is used to create an instance of the component object. Valid values for the argument_tag depend on the component object.

Valid hash object argument tags are

hashexp: n
The hash object’s internal table size, where the size of the hash table is $2^n$.

The value of hashexp is used as a power-of-two exponent to create the hash table size. For example, a value of 4 for hashexp equates to a hash table size of $2^4$, or 16. The maximum value for hashexp is 16, which equates to a hash table size of $2^{16}$ or 65536.

The hash table size is not equal to the number of items that can be stored. Imagine the hash table as an array of ‘buckets.’ A hash table size of 16 would have 16 ‘buckets.’ Each bucket can hold an infinite number of items. The efficiency of the hash table lies in the ability of the hashing function to map items to and retrieve items from the buckets.

You should set the hash table size relative to the amount of data in the hash object in order to maximize the efficiency of the hash object lookup routines. Try different hashexp values until you get the best result. For example, if the hash object contains one million items, a hash table size of 16
(hashexp = 4) would work, but not very efficiently. A hash table size of 512 or 1024 (hashexp = 9 or 10) would result in the best performance.

**Default:** 8, which equates to a hash table size of $2^8$ or 256

**dataset:** 'dataset_name'

Names a SAS data set to load into the hash object. The name of the SAS data set can be a literal or character variable. The data set name must be enclosed in single or double quotation marks. Macro variables must be enclosed in double quotation marks.

**Note:** If the data set contains duplicate keys, then the first instance will be in the hash object, and following instances will be ignored.

**ordered:** 'option'

Specifies whether or how the data is returned in key-value order if you use the hash object with a hash iterator object or if you use the hash object OUTPUT method.

- **option** can be one of the following values:
  - 'ascending' | 'a'  Data is returned in ascending key-value order. Specifying 'ascending' is the same as specifying 'yes'.
  - 'descending' | 'd'  Data is returned in descending key-value order.
  - 'YES' | 'Y'  Data is returned in ascending key-value order. Specifying 'yes' is the same as specifying 'ascending'.
  - 'NO' | 'N'  Data is returned in some undefined order.

**Default:** NO

The argument can also be enclosed in double quotation marks. The valid hash iterator object constructor is the hash object name.

**See Also:** “Initializing Hash Object Data Using a Constructor” and “Declaring and Instantiating a Hash Iterator Object” in *SAS Language Reference: Concepts*.

**Details**

To use a DATA step component object in your SAS program, you must declare and create (instantiate) the object. The DATA step component interface provides a mechanism for accessing the hash and hash iterator predefined component objects from within the DATA step.

If you use the _NEW_ statement to instantiate the component object, you must first use the DECLARE statement to declare the component object. For example, in the following lines of code, the DECLARE statement tells SAS that the variable H is a hash object. The _NEW_ statement creates the hash object and assigns it to the variable H.

```
declare hash h;
    h = _new_ hash( );
```

**Note:** You can use the DECLARE statement to declare and instantiate a DATA step component object in one step.

A constructor is a method that is used to instantiate a component object and to initialize the component object data. For example, in the following lines of code, the _NEW_ statement instantiates a hash object and assigns it to the variable H. In addition, the hash table size is initialized to a value of 16 ($2^4$) using the argument tag, hashexp.

```
declare hash h;
    h = _new_ hash(hashexp: 4);
```
Statements △ _NEW_ Statement 1329

For more information about the predefined DATA step component objects and constructors, see “Using DATA Step Component Objects” in *SAS Language Reference: Concepts*.

**Comparisons**

The _NEW_ statement instantiates a DATA step component object. The DECLARE statement declares a DATA step object.

**Examples**

This example uses the _NEW_ statement to instantiate and initialize data for a hash object and instantiate a hash iterator object.

The hash object is filled with data, and the iterator is used to retrieve the data in key order.

data kennel;
   input name $1-10 kenno $14-15;
datalines;
Charlie 15
Tanner 07
Jake 04
Murphy 01
Pepe 09
Jacques 11
Princess Z 12
;
run;

data _null_;  
   if _N_ = 1 then do;
      length kenno $2;
      length name $10;
      /* Declare the hash object */
      declare hash h;
      /* Instantiate and initialize the hash object */
      h = _new_ hash(hashexp: 4, dataset:"work.kennel", ordered: 'yes');
      /* Declare the hash iterator object */
      declare hiter iter;
      /* Instantiate the hash iterator object */
      iter = _new_ hiter('h');
      /* Define key and data variables */
      h.defineKey('kenno');
      h.defineData('name', 'kenno');
      h.defineDone();
      /* avoid uninitialized variable notes */
      call missing(kenno, name);
   end;
   /* Find the first key in the ordered hash object and output to the log */
   rc = iter.first();
   do while (rc = 0);
      put kenno ' ' name;
      rc = iter.next();
end;
run;

See Also

Statements:
“DECLARE Statement” on page 1135
“Using DATA Step Component Objects” in SAS Language Reference: Concepts
Appendix 1, “DATA Step Object Attributes and Methods,” on page 1635

Null Statement

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;
```
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.

```sas
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
  ;;;;
run;
```

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.

```sas
data _null_; set dsn;
  file print header=header;
  put 'report text';
  ...more statements...
  return;
  header:;
run;
```

See Also

Statements:
  “DATALINES Statement” on page 1133
  “DATALINES4 Statement” on page 1134
  “GO TO Statement” on page 1213
  “LABEL Statement” on page 1280

**ODS CHTML Statement**

Produces a compact, minimal HTML that does not use style information

Valid in: anywhere
Category: ODS: Third-Party Formatted
**ODS _ALL_ CLOSE Statement**

Closes all open ODS output destinations, including the Listing destination

Valid: anywhere

Category: ODS: Output Control


---

**ODS CSVALL Statement**

Produces output that contains columns of data values that are separated by commas. ODS CSVALL produces tabular output with titles, notes, and bylines.

Valid in: anywhere

Category: ODS: Third-Party Formatted


---

**ODS DECIMAL_ALIGN Statement**

Aligns values by the decimal point in numeric columns when no justification is specified

Valid: anywhere

Category: ODS: SAS Formatted


Alias: ODS DECIMAL_ALIGN=YES | ON

Default: YES

Interaction: The ODS DECIMAL_ALIGN statement only effects the RTF destination and the printer family of destinations.

---

**ODS DOCBOOK Statement**

Produces XML output that conforms to the DocBook DTD by OASIS
ODS DOCUMENT Statement

Creates an ODS document

Valid in: anywhere
Category: ODS: SAS Formatted

ODS EXCLUDE Statement

Specifies output objects to exclude from ODS destinations

Valid: anywhere
Category: ODS: Output Control

CAUTION:
After SAS 9 the default destination for ODS HTML will be the current HTML release.
**ODS HTML3 Statement**

Opens, manages, or closes the HTML3 destination, which produces HTML 3.2 formatted output

Valid: anywhere
Category: ODS: Third-Party Formatted

**ODS HTMLCSS Statement**

Produces HTML that is similar to ODS HTML with cascading style sheets

Valid in: anywhere
Category: ODS: Third-Party Formatted

**ODS IMODE Statement**

Produces HTML that is a column of output, separated by lines

Valid in: anywhere
Category: ODS: Third-Party Formatted

**ODS LISTING Statement**

Opens, manages, or closes the LISTING destination

Valid: anywhere
Category: ODS: SAS Formatted
**ODS MARKUP Statement**

Opens, manages, or closes one or more specified destinations

Valid in: anywhere

Category: ODS: Third-Party Formatted


---

**ODS OUTPUT Statement**

Creates a SAS data set from an output object and manages the selection and exclusion lists for the Output destination

Valid: anywhere

Category: ODS: SAS Formatted


---

**ODS PATH Statement**

Specifies which locations to search for definitions that were created by PROC TEMPLATE, as well as the order in which to search for them

Valid: anywhere

Category: ODS: Output Control


---

**ODS PCL Statement**

Provides portable support for PCL (HP LaserJet) files
Valid: anywhere
Category: ODS: Third-Party Formatted

ODS PDF Statement

Opens, manages, or closes the PDF destination
Valid in: anywhere
Category: ODS: Third-Party Formatted

ODS PHTML Statement

Produces a basic HTML that uses twelve style elements and no class attributes
Valid in: anywhere
Category: ODS: Third-Party Formatted

ODS PRINTER Statement

Opens, manages, or closes the PRINTER destination
Valid: anywhere
Category: ODS: Third-Party Formatted

ODS PROCLABEL Statement

Enables you to change a procedure label
ODS PROCTITLE Statement

Determines whether to suppress the writing of the title that identifies the procedure that produces the results

Valid:     anywhere
Category:  ODS: Output Control

ODS PS Statement

Opens, manages, or closes the PostScript destination

Valid:     anywhere
Category:  ODS: Third-Party Formatted

ODS RESULTS Statement

Tracks ODS output in the Results window

Valid:     anywhere
Category:  ODS: Output Control
**ODS RTF Statement**

Creates SAS output for Microsoft Word  
Valid: anywhere  
Category: ODS: Third-Party Formatted  

**ODS SELECT Statement**

Specifies output objects to send to the ODS destinations  
Valid: anywhere  
Category: ODS: Output Control  

**ODS SHOW Statement**

Writes the specified selection or exclusion list to the SAS log  
Valid: anywhere  
Category: ODS: Output Control  

**ODS TRACE Statement**

Writes to the SAS log a record of each output object that is created, or suppresses the writing of this record  
Valid: anywhere  
Category: ODS: Output Control  
**ODS USEGOPT Statement**

Enables graphics option settings  
Valid: anywhere  
Category: ODS: Output Control  

**ODS VERIFY Statement**

Writes or suppresses a warning that a style definition or a table definition that is used is not supplied by SAS  
Valid: anywhere  
Category: ODS: Output Control  

**ODS WML Statement**

Uses the Wireless Application Protocol (WAP) to produce a Wireless Markup Language (WML) DTD with a simple href list for a table of contents  
Valid in: anywhere  
Category: ODS: Third-Party Formatted  

**OPTIONS Statement**

Changes the value of one or more SAS system options
Valid: anywhere
Category: Program Control
See: OPTIONS Statement in the documentation for your operating environment.

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 written to SAS output and sets a line size of 72:

options nodate linesize=72;

See Also
“Definition of System Options” on page 1443
OUTPUT Statement

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 1342 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:

- This line of code writes the current observation to a SAS data set.
  
  ```sas
  output;
  ```

- This line of code writes the current observation to a SAS data set when a specified condition is true.
  
  ```sas
  if deptcode gt 2000 then output;
  ```

- This line of code writes an observation to the data set MARKUP when the PHONE value is missing.
  
  ```sas
  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:

```sas
data response(drop=time1-time3);
  set sulfa;
  time=time1;
  output;
  time=time2;
  output;
```
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:

```sas
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:

```sas
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 Statement” on page 1126  
  “MODIFY Statement” on page 1310  
  “PUT Statement” on page 1344  
  “REMOVE Statement” on page 1377  
  “REPLACE Statement” on page 1381

PAGE Statement

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 Statement” on page 1297
System Options:
  “LINESIZE= System Option” on page 1545
  “PAGESIZE= System Option” on page 1581

PUT Statement

Writes lines to the SAS log, to the SAS output window, or to an external location 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 location, even if the current output line is blank
  □ releases an output line that is being held with a trailing @ by a previous PUT statement.
For an example, see Example 5 on page 1358. For more information, see “Using Line-Hold Specifiers” on page 1352.

**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 1346, but documented more completely in PUT Statement for ODS in *SAS Output Delivery System: User’s Guide*.

*(variable-list)*

specifies a list of variables whose values are written.

**Requirement:** The (format-list) must follow the (variable-list).

**See:** “PUT Statement, Formatted” on page 1362

'*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 1349

**Example:** This statement writes HELLO when the hexadecimal string is converted to ASCII characters:

```sparql
put '68656C6C6F'x;
```

*n*

specifies to repeat n times the subsequent character string.

**Example:** This statement writes a line of 132 underscores.

```sparql
put 132*'_'
```

**Featured in:** Example 4 on page 1358

*pointer-control*

moves the output pointer to a specified line or column in the output buffer.

**See:** “Column Pointer Controls” on page 1346 and “Line Pointer Controls” on page 1348

*column-specifications*

specifies which columns of the output line the values are written.

**See:** “Column Output” on page 1349

**Featured in:** Example 2 on page 1355

*format.*

specifies a format to use when the variable values are written.

**See:** “Formatted Output” on page 1350

**Featured in:** Example 1 on page 1354

*(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 Statement, Formatted” on page 1362

**_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 an automatic variable that references the current INPUT buffer. You can use this automatic 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;
pull _infile_;
```

**Featured in:** Example 6 on page 1359

**_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 1350

**_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 to 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 Statement for ODS” in *SAS Output Delivery System: User’s Guide*

**@|@@**
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:** “Using Line-Hold Specifiers” on page 1352

**Featured in:** Example 5 on page 1358

**Column Pointer Controls**

**@n**
moves the pointer to column n.
Statements △ PUT Statement

Range: a positive integer
Example: @15 moves the pointer to column 15 before the value of NAME is written:

```sas
put @15 name $10.;
```

Featured in: Example 2 on page 1355 and Example 4 on page 1358

@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:

```sas
a=15;
put @a name $10.;
```

Featured in: Example 2 on page 1355

@(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:

```sas
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:

```sas
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 1355

### 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:

```plaintext
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:

```plaintext
put name age / id 3-4;
```

**Featured in:** Example 3 on page 1356

**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 file. Set the N= option in the FILE statement to 1 and direct the PUT statements to a 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:

```plaintext
put @15 ‘Report Title’ overprint @15 ‘____________’;
```

**Featured in:** Example 4 on page 1358

**_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 1356

_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 printed, _PAGE_ produces an output line that contains the appropriate carriage control character. _PAGE_ has no effect on a file that is not printed.

Featured in: Example 3 on page 1356

Details

When to Use PUT
Use the PUT statement to write lines to the SAS log, to the SAS output window, or to an external location. 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 SAS output window.

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 following 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 Statement, Column” on page 1360 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;
```

* The ruled line is for illustrative purposes only; the PUT statement does not generate it.
writes the values for NAME and AGE to the SAS log:*

```
---+----1----+----2----+
 Petson 21
 Morgan 17
```

See “PUT Statement, List” on page 1366 for more information.

**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 enable you to write in a non-standard 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 mmdyy10.;
```

writes the values for NAME, AGE, and DATE to the SAS log:*

```
---+----1----+----2----+
 Petson 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 Statement, Formatted” on page 1362 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=Petson age=21
 name=Morgan age=17
```

See “PUT Statement, Named” on page 1370 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 mmdyy8. ‘ weighs ‘
    startwght +(1) ‘.’ idno= 40-45;
```

See Example 1 on page 1354 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 Location After a Value Is Written” on page 1352.

**Avoiding a Common Error When Writing Both a Character Constant and a Variable**

When using a PUT statement to write a character constant that is followed by a variable name, always put a blank space between the closing quotation mark and the variable name:

```
put ‘Player:’ name1 ‘Player:’ name2 ‘Player:’ name3;
```

* The ruled line is for illustrative purposes only; the PUT statement does not generate it.
Otherwise, SAS might interpret a character constant that is followed by a variable name as a special SAS constant as illustrated in this table.

**Table 7.8** Characters That Cause Misinterpretation When They Follow a Character Constant

<table>
<thead>
<tr>
<th>Character constants followed by a variable starting with</th>
<th>could be interpreted as a...</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>bit testing constant</td>
<td>'00100000'b</td>
</tr>
<tr>
<td>d</td>
<td>date constant</td>
<td>'01jan04'd</td>
</tr>
<tr>
<td>dt</td>
<td>datetime constant</td>
<td>'18jan2003:9:27:05am'dt</td>
</tr>
<tr>
<td>n</td>
<td>name literal</td>
<td>'My Table'n</td>
</tr>
<tr>
<td>t</td>
<td>time constant</td>
<td>'9:25:19pm't</td>
</tr>
<tr>
<td>x</td>
<td>hexadecimal notation</td>
<td>'534153'x</td>
</tr>
</tbody>
</table>

Example 7 on page 1359 shows how to use character constants followed by variables. For more information about SAS name literals and SAS constants in expressions, see *SAS Language Reference: Concepts*.

**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. The following table lists all pointer controls that are available in the PUT statement.

**Table 7.9** Pointer Controls Available in the PUT Statement

<table>
<thead>
<tr>
<th>Pointer Controls</th>
<th>Relative</th>
<th>Absolute</th>
</tr>
</thead>
<tbody>
<tr>
<td>column pointer controls</td>
<td>+n</td>
<td>@n</td>
</tr>
<tr>
<td></td>
<td>+numeric-variable</td>
<td>@numeric-variable</td>
</tr>
<tr>
<td></td>
<td>+(expression)</td>
<td>@(expression)</td>
</tr>
<tr>
<td>line pointer controls</td>
<td>/, <em>PAGE</em>, <em>BLANKPAGE</em></td>
<td>#n</td>
</tr>
<tr>
<td></td>
<td>#numeric-variable</td>
<td>#(expression)</td>
</tr>
</tbody>
</table>
Note: Always specify pointer controls before the variable for which they apply.

See “Pointer Location After a Value Is Written” on page 1352 for more information about 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 “When the Pointer Goes Past the End of a Line” on page 1353.

### 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 located in 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 located 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 located 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 is 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 location.

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 the SAS output window)
- the LRECL= option in the current FILE statement (for external files).

You can 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 Statement” on page 1157.

**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 an external location. SAS allows one-dimensional or multidimensional arrays, but it does not allow a _TEMPORARY_ array. Enclose the subscript in braces, brackets, or parentheses, and print the array using list, formatted, column, 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 location 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 an external location, 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:

```sas
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

<table>
<thead>
<tr>
<th>Name</th>
<th>Style</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME, STARTWGHT</td>
<td>list output</td>
</tr>
<tr>
<td>DATE</td>
<td>formatted output</td>
</tr>
<tr>
<td>IDNO</td>
<td>named output</td>
</tr>
</tbody>
</table>

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 beginning 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 “Pointer Location After a Value Is Written” on page 1352.

* The ruled line is for illustrative purposes only; the PUT statement does not generate it.
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 1368. To remove the unwanted blank that occurs between the value and the period, move the pointer backward by one space.

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:

---

* The ruled line is for illustrative purposes only; the PUT statement does not generate it.
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 step 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 to 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 of metropolitan and non-metropolitan 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 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 .1 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 .3160 1.060 2
 OK 1.870 1.276 2 TX 14.166 2.821 2
 OH 8.826 2.021 3 IN .396 .1582 3
 IL 9.574 1.857 3 WI 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

** The ruled line is for illustrative purposes only; the PUT statement does not generate it.
Statements △ PUT Statement 1357

```
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 7.23  PUT Statement Output for the Northeast Region

```
<table>
<thead>
<tr>
<th>State</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>ME</td>
<td>1.228 million</td>
</tr>
<tr>
<td>NH</td>
<td>1.109 million</td>
</tr>
<tr>
<td>VT</td>
<td>0.563 million</td>
</tr>
<tr>
<td>MA</td>
<td>6.017 million</td>
</tr>
<tr>
<td>RI</td>
<td>1.003 million</td>
</tr>
<tr>
<td>CT</td>
<td>3.288 million</td>
</tr>
<tr>
<td>NY</td>
<td>17.990 million</td>
</tr>
<tr>
<td>NJ</td>
<td>7.730 million</td>
</tr>
<tr>
<td>PA</td>
<td>11.882 million</td>
</tr>
</tbody>
</table>

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:

```sas
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:

```sas
  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:

```sas
data _null_;
  input idno name $ startwght 3.;
  put name @;
  if startwght ne . then
    put @15 startwght;
  else put;
data _null_;
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
```

* The ruled line is for illustrative purposes only; the PUT statement does not generate it.
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.

```sas
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 line to the SAS log:

```
----+----1----+----2
219 red 210 192
```

Example 7: Avoiding a Common Error When Writing a Character Constant Followed by a Variable

This example illustrates how to use a PUT statement to write character constants and variable values without causing them to be misinterpreted as SAS name literals. A SAS name literal is a name token that is expressed as a string within quotation marks, followed by the letter n. For more information about SAS name literals, see SAS Language Reference: Concepts.

In the program below, the PUT statement writes the constant ‘n’ followed by the value of the variable NVAR1, and then writes another constant ‘n’:

```sas
data _null_;  
  n=5;  
  nvvar1=1;  
  var1=7;  
  put @1 ‘n’ nvvar1 ‘n’;  
run;

This program writes the following line to the SAS log:

```
----+----1----+----2
n1 n
```

If all the spaces between the constants and the variables are removed from the previous PUT statement, SAS interprets ‘n’n as a name literal instead of reading ‘n’ as a constant. The next variable is read as VAR1 instead of NVAR1. The final ‘n’ constant is interpreted correctly.

```sas
put @1 ‘n’nvar1’n’;
```

* The ruled line is for illustrative purposes only; the PUT statement does not generate it.
This PUT statement writes the following line to the SAS log:

```
-----+----1------+-+++++2
 5 7  n
```

To print character constants and variable values without intervening spaces, and without potential misinterpretation, you can add spaces between them and use pointer controls where necessary. For example, the following PUT statement uses a pointer control to write the correct character constants and variable values but does not insert blank spaces. Note that +(-1) moves the PUT statement pointer backwards by 1 space.

```
put @1 'n' nvar1 +(-1) 'n';
```

This PUT statement writes the following line to the SAS log:

```
-----+----1------+-+++++2
 n1 n
```

**See Also**

Statements:
- “FILE Statement” on page 1157
- “PUT Statement, Column” on page 1360
- “PUT Statement, Formatted” on page 1362
- “PUT Statement, List” on page 1366
- “PUT Statement, Named” on page 1370
- PUT Statement for ODS

System Options:
- “LINESIZE= System Option” on page 1545
- “PAGESIZE= System Option” on page 1581

---

**PUT Statement, 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> <@ | @@>;
```

* The ruled line is for illustrative purposes only; the PUT statement does not generate it.
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 1361

@ | @@
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 1352

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 “Using Multiple Output Styles in a Single PUT Statement” on page 1350.

Examples

Use column output in the PUT statement as shown here.

□ This PUT statement uses column output:

```plaintext
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  ```
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:

```
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 Statement” on page 1344
Arguments

**pointer-control**

moves the output pointer to a specified line or column.

**See:** “Column Pointer Controls” on page 1346 and “Line Pointer Controls” on page 1348

**Featured in:** Example 1 on page 1365

**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 1364

**Featured in:** Example 1 on page 1365

**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 1365

**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 67.

**pointer-control**

specifies one of these pointer controls to use to position a value: @, #, /, +, and OVERPRINT.

**Example:** Example 1 on page 1365

**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 1365

\(n^*\) 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:

```sas
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 1364

@ | @@ 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 1352

### 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 of Formats” on page 71.

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:

```sas
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:

```sas
put (score1-score5) (4. 4. 4. 4. 4.);
```

A shorter version of the previous statement is

```sas
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 an example, see Example 3 on page 1365.

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
Statements △ PUT Statement, Formatted 1365

 Температурный 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 “Arrays” on page 1353.

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:

    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.)

Example 3: Including More Format Specifications Than Necessary  This format list
includes more specifications than are necessary when the PUT statement executes:

    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: *

```
----+----1----+
2 24 36
0 20 30
```

See Also

Statement:

“PUT Statement” on page 1344

---

**PUT Statement, 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 1346 and “Line Pointer Controls” on page 1348

**Featured in:** Example 2 on page 1369

**variable**

names the variable whose value is written.

**Featured in:** Example 1 on page 1369

* The ruled line is for illustrative purposes only; the PUT statement does not generate it.
\[ n^* \]

specifies to repeat \( n \) times the subsequent character string.

**Example:** This statement writes a line of 132 underscores:

```plaintext
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 “When the Pointer Goes Past the End of a Line” on page 1353.

**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 1368

**Featured in:** Example 2 on page 1369

: enables you to specify a format that the PUT statement uses to write the variable value. All leading and trailing blanks are 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 1368

**Featured in:** Example 3 on page 1370

- enables you to specify a format that the PUT statement uses to write the variable value. SAS displays the formatted value in quotation marks even if the formatted value does not contain the delimiter. SAS deletes all leading and trailing blanks, and each value is followed by a single blank. Missing values for character variables are written as a blank (" ") and, by default, missing values for numeric variables are written as a period (".").

**Requirement:** You must specify the DSD option in the FILE statement.

**Featured in:** Example 4 on page 1370

`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 67.

**Featured in:** Example 3 on page 1370

`@` 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 1352

### 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 1370.

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 might 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 1369.

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:

```sas
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:

```
2,353.20 7.10
6,231.00 121.00
```

In comparison, the following example uses formatted output:

```sas
put x comma10.2 y 7.2;
```
These lines are written to the SAS log, with the values aligned in columns:

----+----1----+----2----+----3----+----4
2,353.20  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 $ startwgght;
  put name ‘weighs ’ startwgght +(-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 using the : argument:

```sas
data _null_;  
input salesrep : $10. tot : comma6. date : date9.;  
put Week of date : worddate15.  
salesrep : $12. 'sales were '  
tot : dollar9. + (-1) '.';  
datalines;  
Wong 15,300 12OCT2004  
Hoffman 9,600 12OCT2004  
;
```

These lines appear in the SAS log:

- Week of Oct 12, 2004 Wong sales were $15,300.
- Week of Oct 12, 2004 Hoffman sales were $9,600.

Example 4: Writing Values with Modified List Output and ~  This DATA step uses modified list output to write several variable values in the output line using the ~ argument:

```sas
data _null_;  
input salesrep : $10. tot : comma6. date : date9.;  
file log delimiter=" " dsd;  
put Week of date ~ worddate15.  
salesrep ~ $12. 'sales were '  
tot ~ dollar9. + (-1) '.';  
datalines;  
Wong 15,300 12OCT2004  
Hoffman 9,600 12OCT2004  
;
```

These lines appear in the SAS log:

- Week of "Oct 12, 2004" "Wong" sales were "$15,300".
- Week of "Oct 12, 2004" "Hoffman" sales were "$9,600".

See Also

Statements:

- “PUT Statement” on page 1344
- “PUT Statement, Formatted” on page 1362

---

**PUT Statement, 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 1346 and “Line Pointer Controls” on page 1348

**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:

```plaintext
put x= dollar7.2;
```

When X=100, the formatted value uses seven columns:

```plaintext
X=$100.00
```

See: “Formatting Named Output” on page 1372

**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 Specifications” on page 1352
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 of Formats” on page 71.

Examples

Use named output in the PUT statement as shown here.

□ This PUT combines named output with column pointer controls to align the output:

```sas
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:

```sas
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:

```sas
----+----1----+----2----+----3----+----4
ITEM=binders AMOUNT=$153.25
```

See Also

Statement:

“PUT Statement” on page 1344
PUT, ODS Statement

Writes data values to a special buffer from which they can be written to the data component and formatted by ODS

Valid: in a DATA step
Category: File-handling
Type: Executable
See Also: The PUT, ODS statement in SAS Output Delivery System: User’s Guide

PUTLOG Statement

Writes a message to the SAS log

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

Syntax

PUTLOG 'message';

Arguments

message
specifies the message that you want to write to the SAS log. Message can include character literals (enclosed in quotation marks), variable names, formats, and pointer controls.

Tip: You can precede your message text with WARNING, MESSAGE, or NOTE to better identify the output in the log.

Details

The PUTLOG statement writes a message that you specify to the SAS log.

The PUTLOG statement is helpful when you use macro-generated code because you can send output to the SAS log without affecting the current file destination.

Comparisons

The PUTLOG statement is similar to the ERROR statement except that PUTLOG does not set _ERROR_ to 1.

Examples

Example 1: Writing Messages to the SAS Log Using the PUTLOG Statement

The following program creates the computeAverage92 macro, which computes the average score, validates input data, and uses the PUTLOG statement to write error messages to
the SAS log. The DATA step uses the PUTLOG statement to write a warning message to the log.

```sas
data ExamScores;
  input Name $ 1-16 Score1 Score2 Score3;
  datalines;
  Sullivan, James  86 92 88
  Martinez, Maria  95 91 92
  Guzik, Eugene    99 98 .
  Schultz, John    90 87 93
  van Dyke, Sylvia 98 . 91
  Tan, Carol       93 85 85
;
  options pageno=1 nodate linesize=80 pagesize=60;
  filename outfile 'your-output-file';
  /* Create a macro that computes the average score, validates input data, and uses PUTLOG to write error messages to the SAS log. */
  %macro computeAverage92(s1, s2, s3, avg);
  if &s1 < 0 or &s2 < 0 or &s3 < 0 then do;
    putlog 'ERROR: Invalid score data ' &s1= &s2= &s3=;
    &avg = .;
  end;
  else
    &avg = mean(&s1, &s2, &s3);
%mend;

  data _null_;
  set ExamScores;
  file outfile;
  %computeAverage92(Score1, Score2, Score3, AverageScore);
  put name Score1 Score2 Score3 AverageScore;

  /* Use PUTLOG to write a warning message to the SAS log. */
  if AverageScore < 92 then
    putlog 'WARNING: Score below the minimum ' name= AverageScore= 5.2;
  run;

  proc print;
  run;
```

The following lines are written to the SAS log.
Output 7.24  SAS Log Results from the PUTLOG Statement

| WARNING: Score below the minimum Name=Sullivan, James AverageScore=88.67 |
| ERROR: Invalid score data Score1=99 Score2=98 Score3=. |
| WARNING: Score below the minimum Name=Guzik, Eugene AverageScore=. |
| WARNING: Score below the minimum Name=Schultz, John AverageScore=90.00 |
| ERROR: Invalid score data Score1=98 Score2=. Score3=91 |
| WARNING: Score below the minimum Name=van Dyke, Sylvia AverageScore=. |
| WARNING: Score below the minimum Name=Tan, Carol AverageScore=87.67 |

SAS creates the following output file.

Output 7.25  Individual Examination Scores

<table>
<thead>
<tr>
<th>Exam Scores</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obs</td>
<td>Name</td>
</tr>
<tr>
<td>1</td>
<td>Sullivan, James</td>
</tr>
<tr>
<td>2</td>
<td>Martinez, Maria</td>
</tr>
<tr>
<td>3</td>
<td>Guzik, Eugene</td>
</tr>
<tr>
<td>4</td>
<td>Schultz, John</td>
</tr>
<tr>
<td>5</td>
<td>van Dyke, Sylvia</td>
</tr>
<tr>
<td>6</td>
<td>Tan, Carol</td>
</tr>
</tbody>
</table>

See Also

Statement:

“ERROR Statement” on page 1155

REDIRECT Statement

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 “Stored Compiled DATA Step Programs” in *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.

```sas
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 Statement” on page 1126
“Stored Compiled DATA Step Programs” in SAS Language Reference: Concepts

REMOVE Statement

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.

```sas
libname perm 'SAS-data-library';

data perm.accounts;
   input AcctNumber Credit;
   datalines;
1001 1500
1002 4900
1003 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:

<table>
<thead>
<tr>
<th>OBS</th>
<th>Acct Number</th>
<th>Credit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1001</td>
<td>1500</td>
</tr>
<tr>
<td>3</td>
<td>1003</td>
<td>3000</td>
</tr>
</tbody>
</table>

**See Also**

Statements:

“DELETE Statement” on page 1140
“IF Statement, Subsetting” on page 1214
“MODIFY Statement” on page 1310
“OUTPUT Statement” on page 1341
“REPLACE Statement” on page 1381
RENAME Statement

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= Data Set Option” on page 45
REPLACE Statement

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;
```
RETAIN Statement

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 also use a shorthand notation for specifying a range of sequential integers. The increment is always +1.

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
Statements △ RETAIN Statement

- a variable whose value is assigned in a sum statement
- the automatic variables _N_, _ERROR_, _L_, _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_;
  ```

- All of these statements assign initial values of 1 through 4 to VAR1 through VAR4:
  
  ```
  □ retain var1-var4 (1 2 3 4);
  □ retain var1-var4 (1,2,3,4);
  □ retain var1-var4(1:4);
  ```
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:

```plaintext
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_=1
```

```
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=10000 i=4  
_ERROR_=0 _N_=2
```

Values at beginning of DATA step:
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 Statement” on page 1110
“BY Statement” on page 1114
“INPUT Statement” on page 1248

RETURN Statement

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 Statement” on page 1157
  “GO TO Statement” on page 1213
  “LINK Statement” on page 1296
RUN Statement

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.
The CANCEL option has no effect when you use the KILL option with PROC DATASETS.

Details
Although 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;
SAS writes the following message to the log:

WARNING: DATA step not executed at user’s request.

### %RUN Statement

**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 Statement” on page 1218
  “RUN Statement” on page 1389

SASFILE Statement

Opens a SAS data set and allocates enough buffers to hold the entire file in memory

Valid:    Anywhere
Category:  Program Control
Restriction:  A SAS data set opened by the SASFILE statement can be used for subsequent input (read) or update processing but not for output or utility processing.
See:     SASFILE Statement in the documentation for your operating environment.

Syntax

SASFILE <libref>.member-name.<member-type> <(password-option(s))> OPEN | LOAD | CLOSE ;

Arguments

libref

a name that is associated with a SAS data library. The libref (library reference) must be a valid SAS name. The default libref is either USER (if assigned) or WORK (if USER not assigned).

Restriction:  The libref cannot represent a concatenation of SAS data libraries that contain a library in sequential format.

member-name

a valid SAS name that is a SAS data file (a SAS data set with the member type DATA) that is a member of the SAS data library associated with the libref.

Restriction:  The SAS data set must have been created with the V7, V8, or V9 Base SAS engine.

member-type

the type of SAS file to be opened. Valid value is DATA, which is the default.

password-option(s)

specifies one or more of the following password options:

READ=password

enables the SASFILE statement to open a read-protected file. The password must be a valid SAS name.

WRITE=password

enables the SASFILE statement to use the write password to open a file that is both read-protected and write-protected. The password must be a valid SAS name.
ALTER=password enables the SASFILE statement to use the alter password to open a file that is both read-protected and alter-protected. The password must be a valid SAS name.

PW=password enables the SASFILE statement to use the password to open a file that is assigned for all levels of protection. The password must be a valid SAS name.

Tip: When SASFILE is executed, SAS checks whether the file is read-protected. Therefore, if the file is read-protected, you must include the READ= password in the SASFILE statement. If the file is either write-protected or alter-protected, you can use a WRITE=, ALTER=, or PW= password. However, the file is opened only in input (read) mode. For subsequent processing, you must specify the necessary password(s). See Example 2 on page 1395.

OPEN opens the file, allocates the buffers, but defers reading the data into memory until a procedure, statement, or application is executed.

LOAD opens the file, allocates the buffers, and reads the data into memory.

Note: If the total number of allowed buffers is less than the number of buffers required for the file based on the number of data set pages and index file pages, SAS issues a warning to tell you how many pages are read into memory.

CLOSE frees the buffers and closes the file.

Details

General Information The SASFILE statement opens a SAS data set and allocates enough buffers to hold the entire file in memory. Once it is read, data is held in memory, available to subsequent DATA and PROC steps or applications, until either a second SASFILE statement closes the file and frees the buffers or the program ends, which automatically closes the file and frees the buffers.

Using the SASFILE statement can improve performance by

- reducing multiple open/close operations (including allocation and freeing of memory for buffers) to process a SAS data set to one open/close operation
- reducing I/O processing by holding the data in memory.

If your SAS program consists of steps that read a SAS data set multiple times and you have an adequate amount of memory so that the entire file can be held in real memory, the program should benefit from using the SASFILE statement. Also, SASFILE is especially useful as part of a program that starts a SAS server such as a SAS/SHARE server. However, as with most performance-improvement features, it is suggested that you set up a test in your environment to measure performance with and without the SASFILE statement.

Processing a SAS Data Set Opened with SASFILE When the SASFILE statement executes, SAS opens the specified file. Then when subsequent DATA and PROC steps execute, SAS does not have to open the file for each request; the file remains open until a second SASFILE statement closes it or the program or session ends.

When a SAS data set is opened by the SASFILE statement, the file is opened for input processing and can be used for subsequent input or update processing. However, the file cannot be used for subsequent utility or output processing, because utility and output processing requires exclusive access to the file (member-level locking). For example, you cannot replace the file or rename its variables.
Table 7.10 on page 1393 provides a list of some SAS procedures and statements and specifies whether they are allowed if the file is opened by the SASFILE statement:

<table>
<thead>
<tr>
<th>Processing Request</th>
<th>Open Mode</th>
<th>Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPEND procedure</td>
<td>update</td>
<td>Yes</td>
</tr>
<tr>
<td>DATA step that creates or replaces the file</td>
<td>output</td>
<td>No</td>
</tr>
<tr>
<td>DATASETS procedure to rename or add a variable, add or change a label, or add or remove integrity constraints or indexes</td>
<td>utility</td>
<td>No</td>
</tr>
<tr>
<td>DATASETS procedure with AGE, CHANGE, or DELETE statements</td>
<td>does not open the file but requires exclusive access</td>
<td>No</td>
</tr>
<tr>
<td>FSEDIT procedure</td>
<td>update</td>
<td>Yes</td>
</tr>
<tr>
<td>PRINT procedure</td>
<td>input</td>
<td>Yes</td>
</tr>
<tr>
<td>SORT procedure that replaces original data set with sorted one</td>
<td>output</td>
<td>No</td>
</tr>
<tr>
<td>SQL procedure to modify, add, or delete observations</td>
<td>update</td>
<td>Yes</td>
</tr>
<tr>
<td>SQL procedure with CREATE TABLE or CREATE VIEW statement</td>
<td>output</td>
<td>No</td>
</tr>
<tr>
<td>SQL procedure to create or remove integrity constraints or indexes</td>
<td>utility</td>
<td>No</td>
</tr>
</tbody>
</table>

**Buffer Allocation** A buffer is a reserved area of memory that holds a segment of data while it is processed. The number of allocated buffers determines how much data can be held in memory at one time.

The number of buffers is not a permanent attribute of a SAS file; that is, it is valid only for the current SAS session or job. When a SAS file is opened, a default number of buffers for processing the file is set. The default depends on the operating environment but typically is a small number, for example, one buffer. To specify a different number of buffers, you can use the BUFNO= data set option or system option.

When the SASFILE statement is executed, SAS automatically allocates the number of buffers based on the number of data set pages and index file pages (if an index file exists). For example:

- If the number of data set pages is five and there is not an index file, SAS allocates five buffers.
- If the number of data set pages is 500 and the number of index file pages is 200, SAS allocates 700 buffers.

If a file that is held in memory increases in size during processing, the number of allocated buffers increases to accommodate the file. Note that if SASFILE is executed for a SAS data set, the BUFNO= option is ignored.
I/O Processing  An I/O (input/output) request reads a segment of data from a storage device (such as disk) and transfers the data to memory, or conversely transfers the data from memory and writes it to the storage device. When a SAS data set is opened by the SASFILE statement, data is read once and held in memory, which should reduce the number of I/O requests.

CAUTION:
I/O processing can be reduced only if there is sufficient real memory. If the SAS data set is very large, you may not have sufficient real memory to hold the entire file. If this occurs, your operating environment may simulate more memory than actually exists, which is virtual memory. If virtual memory occurs, data access I/O requests are replaced with swapping I/O requests, which could result in no performance improvement. In addition, both SAS and your operating environment have a maximum amount of memory that can be allocated, which could be exceeded by the needs of your program. If this occurs, the number of allocated buffers may be decreased to the default allocation in order to free memory.

Tip: To determine how much memory a SAS data set requires, execute the CONTENTS procedure for the file to list its page size, the number of data set pages, the index file size, and the number of index file pages.

Using the SASFILE Statement in a SAS/SHARE Environment  The following are considerations for using the SASFILE statement with SAS/SHARE software:

- You must execute the SASFILE statement before you execute the PROC SERVER statement.
- If the client (the computer on which you use a SAS session to access a SAS/SHARE server) executes the SASFILE statement, it is rejected.
- Once the SASFILE statement is executed, all users who subsequently open the file will access the data held in memory instead of data that is stored on the disk.
- Once the SASFILE statement is executed, you cannot close the file and free the buffers until the SAS/SHARE server is terminated.
- You can use the ALLOCATE SASFILE command for the PROC SERVER statement as an alternative that brings part of the file into memory (controlled by the BUFNO= option).
- If the SASFILE statement is executed and you execute ALLOCATE SASFILE specifying a value for BUFNO= that is a larger number of buffers than allocated by SASFILE, performance will not be improved.

Comparisons

- Use the BUFNO= system option or data set option to specify a specific number of buffers.
- With SAS/SHARE software, you can use the ALLOCATE SASFILE command for the PROC SERVER statement to bring part of the file into memory (controlled by the BUFNO= option).

Examples

Example 1: Using SASFILE in a Program with Multiple Steps  The following SAS program illustrates the process of opening a SAS data set, transferring its data to memory, and reading that data held in memory for multiple tasks. The program is composed of steps that read the file multiple times.

libname mydata 'SAS-data-library';
sasfile mydata.census.data open; ①

data test1;
   set mydata.census;  ②
run;

data test2;
   set mydata.census;  ③
run;

proc summary data=mydata.census print;  ④
run;

data mydata.census;  ⑤
   modify mydata.census;
   .
   . (statements to modify data)
   .
run;

sasfile mydata.census close;  ⑥

1 Opens SAS data set MYDATA.CENSUS, and allocates the number of buffers based on the number of data set pages and index file pages.
2 Reads all pages of MYDATA.CENSUS, and transfers all data from disk to memory.
3 Reads MYDATA.CENSUS a second time, but this time from memory without additional I/O requests.
4 Reads MYDATA.CENSUS a third time, again from memory without additional I/O requests.
5 Reads MYDATA.CENSUS a fourth time, again from memory without additional I/O requests. If the MODIFY statement successfully changes data in memory, the changed data is transferred from memory to disk at the end of the DATA step.
6 Closes MYDATA.CENSUS, and frees allocated buffers.

Example 2: Specifying Passwords with the SASFILE Statement  The following SAS program illustrates using the SASFILE statement and specifying passwords for a SAS data set that is both read-protected and alter-protected:

libname mydata 'SAS-data-data-library';
sasfile mydata.census (read=gizmo) open; ①

proc print data=mydata.census (read=gizmo); ②
run;

data mydata.census;
   modify mydata.census (alter=luke); ③
   .
   . (statements to modify data)
   .
run;

1 The SASFILE statement specifies the read password, which is sufficient to open the file.
In the PRINT procedure, the read password must be specified again.

The alter password is used in the MODIFY statement, because the data set is being updated.

*Note:* It is acceptable to use the higher-level alter password instead of the read password in the above example. △

**See Also**

Data Set Option:
- “BUFNO= Data Set Option” on page 10

System Option:
- “BUFNO= System Option” on page 1482

---

### SELECT Statement

**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.

*See:* “Evaluating the when-expression When a select-expression Is Included” on page 1397

**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.
See: “Evaluating the when-expression When a select-expression Is Not Included” on page 1397

**statement**

can be any executable SAS statement, including DO, SELECT, and null statements. You must specify the `statement` argument.

**Details**

**Using WHEN Statements in a SELECT Group**

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.

**Evaluating the when-expression When a select-expression Is Included**

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.

**Evaluating the when-expression When a select-expression Is Not Included**

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.

**Processing Large Amounts of Data with %INCLUDE Files**

One way to process large amounts of data is to use %INCLUDE statements in your DATA step. Using %INCLUDE statements enables you to perform complex processing while keeping your main program manageable. The %INCLUDE files that you use in your main program can contain WHEN statements and other SAS statements to process your data. See Example 5 on page 1399 for an example.

**Comparisons**

Use IF-THEN/ELSE statements for programs with few statements. 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

Example 1: Using Statements

```sql
select (a);
  when (1) x=x*10;
  when (2);
  when (3,4,5) x=x*100;
  otherwise;
end;
```

Example 2: Using DO Groups

```sql
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

```sql
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

```sql
/* 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;
```
Example 5: Processing Large Amounts of Data  In the following example, the %INCLUDE statements contain code that includes WHEN statements to process new and old items in the inventory. The main program shows the overall logic of the DATA step.

data test (keep=ItemNumber);
   set ItemList;
   select;
      %include NewItems;
      %include OldItems;
   otherwise put 'Item ' ItemNumber ' is not in the inventory.';
   end;
run;

See Also

Statements:
   “DO Statement” on page 1145
   “IF Statement, Subsetting” on page 1214
   “IF-THEN/ELSE Statement” on page 1216

SET Statement

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 “SAS Data Sets” in SAS Language Reference: Concepts for a description of the levels of SAS data set names and when to use each level.
specifies actions SAS is to take when it reads variables or observations into the program data vector for processing.

**See:** Refer to “Definition of Data Set Options” on page 6 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 1405

**KEY=index</UNIQUE>**
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 _DSENON and the automatic variable _ERROR_ is set to 1.

**Featured in:** Example 7 on page 1405 and Example 8 on page 1405.

**See Also:** For more information, see the description of the autocall macro SYSRC in *SAS Macro Language: Reference*.

**See Also:** UNIQUE option on page 1402

**CAUTION:**
Continuous loops can occur when you use the KEY= option. If you use the KEY= option without specifying the primary data set, you must include either a STOP statement to stop DATA step processing, or programming logic that uses the _IORC_ automatic variable in conjunction with the SYSRC autocall macro and checks for an invalid value of the _IORC_ variable, or both.

**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 1405

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 previously 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 1404 and Example 9 on page 1405

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 and must be preceded by a slash.

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 1405

See Also: For extensive examples, see 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 1404 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 1403 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 1404 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 1404, Example 7 on page 1405, and Example 8 on page 1405 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:

```sas
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:

```sas
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:

```sas
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 existing 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:

```sas
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:

```sas
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:

```sas
data south;
  set revenue;
  if region=4;
  set expense point=_n_;
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:

```sas
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:

```sas
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:

```sas
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:

```sas
  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
set rental end=last;
totdays + days;
if last then
do;
   revenue=totdays*65.78;
   output;
end;

See Also

Statements:
“BY Statement” on page 1114
“DO Statement” on page 1145
“INPUT Statement” on page 1248
“MERGE Statement” on page 1306
“STOP Statement” on page 1407
“UPDATE Statement” on page 1416

“Rules for Words and Names” in SAS Language Reference: Concepts
“Reading, Modifying, and Combining SAS Data Sets” in SAS Language Reference: Concepts
“Definition of Data Set Options” on page 6

SAS Macro Language: Reference
Combining and Modifying SAS Data Sets: Examples

---

**SKIP Statement**

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 
modes of operation.

### See Also

**Statement:**

“PAGE Statement” on page 1343  

**System Options:**

“LINESIZE= System Option” on page 1545  
“PAGESIZE= System Option” on page 1581

---

**STOP Statement**

**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

```latex
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:

```latex
data sample;
do sampleobs=1 to totalobs by 10;
  set master.research point=sampleobs
      nobs=totalobs;
    output;
end;
stop;
run;
```

See Also

Statements:
“ABORT Statement” on page 1100
POINT= option in the SET statement on page 1401

---

Sum Statement

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 Function” on page 844
Statement:
“RETAIN Statement” on page 1383

TITLE Statement

Specifies title lines for SAS output
Valid: anywhere
Category: Output Control
See: TITLE Statement in the documentation for your operating environment.
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.

You can customize titles by inserting BY variable values (#BYVALn), BY variable names (#BYVARn), 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.

#BYVALn | #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.

Follow these rules when you use #BYVAL in the TITLE statement of a PROC step:

- Specify the variable that is used by #BYVAL in the BY statement.
- Insert #BYVAL in the specified title text string at the position where you want the substitution text to appear.
- Follow #BYVAL 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 #BYVAL substitution to be followed immediately by other text, with no delimiter, use a trailing dot (as with macro variables).

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.

#BYVARn | #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.

Follow these rules when you use #BYVAR in the TITLE statement of a PROC step:

- Specify the variable that is used by #BYVAR in the BY statement.
- Insert #BYVAR in the specified title text string at the position where you want the substitution text to appear.
- Follow #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 substitution to be followed immediately by other text, with no delimiter, use a trailing dot (as with macro variables).

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 System Option” on page 1486.

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 “Expressions” 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.
Comparisons
You can also create titles with the TITLES window.

Examples

Example 1: Using the TITLE Statement  
The following examples show how you can use the TITLE statement:

- This statement suppresses a title on line n and all lines after it:
  ```
  title;
  ```

- These are examples of TITLE statements:
  ```
  title 'First Draft';
  title2 "Year's End Report";
  title2 'Year''s End Report';
  ```

Example 2: Customizing Titles by Using BY Variable Values  
You can customize titles by inserting BY variable values in the titles that you specify in PROC steps. The following examples show how to use #BYVALn, #BYVARn, and #BYLINE:

- title 'Quarterly Sales for #byval(site)';
- title 'Annual Costs for #byvar2';
- title 'Data Group #byline';

Example 3: Customizing Titles and Footnotes by Using the Output Delivery System  
You can customize titles and footnotes with ODS. The following example shows you how to use PROC TEMPLATE to change the color, justification, and size of the text for the title and footnote.

```/*
*The following program creates the data set *
*grain_production and the $cntry format. *
***********************************************************************/
data grain_production;
  length Country $ 3 Type $ 5;
  input Year country $ type $ Kilotons;
datalines;
1995 BRZ Wheat 1516
1995 BRZ Rice 11236
1995 BRZ Corn 36276
1995 CHN Wheat 102207
1995 CHN Rice 185226
1995 CHN Corn 112331
1995 IND Wheat 63007
1995 IND Rice 122372
1995 IND Corn 9800
1995 INS Wheat .
1995 INS Rice 49860
1995 INS Corn 8223
1995 USA Wheat 59494
1995 USA Rice 7888
1995 USA Corn 187300
1996 BRZ Wheat 3302*/```
1996 BRZ Rice  10035
1996 BRZ Corn  31975
1996 CHN Wheat 109000
1996 CHN Rice   190100
1996 CHN Corn  119350
1996 IND Wheat  62620
1996 IND Rice   120012
1996 IND Corn   8660
1996 INS Wheat  .
1996 INS Rice   51165
1996 INS Corn   8925
1996 USA Wheat  62099
1996 USA Rice   7771
1996 USA Corn   236064
;
run;

proc format;
  value $cntry 'BRZ'='Brazil'
     'CHN'='China'
     'IND'='India'
     'INS'='Indonesia'
     'USA'='United States';
run;

/*****************************************
*This PROC TEMPLATE step creates the   *
*table definition TABLE1 that is used   *
in the DATA step.                       *
*****************************************/
proc template;
  define table table1;
    mvar sysdate9;
    dynamic colhd;
    classlevels=on;
  define column char_var;
    generic=on;
    blank_dups=on;
    header=colhd;
    style=cellcontents;
  end;

  define column num_var;
    generic=on;
    header=colhd;
    style=cellcontents;
  end;

  define footer table_footer;
    end;
end;
run;
The ODS LISTING CLOSE statement closes the Listing destination to conserve resources. The ODS HTML statement creates HTML output created with the style definition D3D. The TITLE statement specifies the text for the first title and the attributes that ODS uses to modify it. The J= style attribute left justifies the title. The COLOR= style attributes change the color of the title text "Leading Grain" to blue and "Producers in" to green. The TITLE2 statement specifies the text for the second title and the attributes that ODS uses to modify it. The J= style attribute center justifies the title. The COLOR= attribute changes the color of the title text "1996" to red. The height= attributes change the size of each individual number in "1996". The FOOTNOTE statement specifies the text for the first footnote and the attributes that ODS uses to modify it. The J=left style attribute left justifies the footnote. The height=20 style attribute changes the font size to 20pt. The COLOR= style attributes change the color of the footnote text "Prepared" to red and "on" to green. The FOOTNOTE2 statement specifies the text for the second footnote and the attributes that ODS uses to modify it. The J= style attribute centers the footnote. The COLOR= attribute changes the color of the date to blue. The height= attribute changes the font size of the date specified by the sysdate9 macro. 

/*******************************************************/
ods listing close;
ods html body='newstyle-body.htm'
   style=d3d;
title j=left
   color=blue "Leading Grain"
   color=green "Producers in";
title2 j=center color=red
   height=28pt "1"
   height=24pt "9"
   height=20pt "9"
   height=16pt "6";
footnote j=left height=20pt
   color=red "Prepared"
   color=green "on";
data _null_;  
set grain_production;  
where type in ('Rice', 'Corn') and year=1996;  
file print ods=(  
    template='table1'  
    columns=(  
        char_var=country(generic=on format=$cntry.  
            dynamic=(colhd='Country'))  
        char_var=type(generic dynamic=(colhd='Year'))  
        num_var=kilotons(generic=on format=comma12.  
            dynamic=(colhd='Kilotons'))  
    )  
    );  
put _ods_;  
run;  
ods html close;  
ods listing;
See Also

Statement:
  “FOOTNOTE Statement” on page 1208
System Option:
  “LINESIZE= System Option” on page 1545

UPDATE Statement

Updates a master file by applying transactions

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

Syntax

   <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.
(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 1418
  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, Combining, and Modifying 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 Statement” on page 1308.

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:

```sas
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

<table>
<thead>
<tr>
<th>OBS</th>
<th>ID</th>
<th>NAME</th>
<th>TEAM</th>
<th>WEIGHT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1114</td>
<td>sally</td>
<td>blue</td>
<td>125</td>
</tr>
<tr>
<td>2</td>
<td>1441</td>
<td>sue</td>
<td>green</td>
<td>145</td>
</tr>
<tr>
<td>3</td>
<td>1750</td>
<td>joey</td>
<td>red</td>
<td>189</td>
</tr>
<tr>
<td>4</td>
<td>1994</td>
<td>mark</td>
<td>yellow</td>
<td>165</td>
</tr>
<tr>
<td>5</td>
<td>2304</td>
<td>joe</td>
<td>red</td>
<td>170</td>
</tr>
</tbody>
</table>
Transaction Data Set
FITNESS

<table>
<thead>
<tr>
<th>OBS</th>
<th>ID</th>
<th>NAME</th>
<th>TEAM</th>
<th>WEIGHT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1114</td>
<td>sally</td>
<td>blue</td>
<td>119</td>
</tr>
<tr>
<td>2</td>
<td>1994</td>
<td>mark</td>
<td>yellow</td>
<td>174</td>
</tr>
<tr>
<td>3</td>
<td>2304</td>
<td>joe</td>
<td>red</td>
<td>170</td>
</tr>
</tbody>
</table>

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 7.26

<table>
<thead>
<tr>
<th>Weekly Weigh-in Report</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBS</td>
<td>STATUS</td>
</tr>
<tr>
<td>-----</td>
<td>--------</td>
</tr>
<tr>
<td>1</td>
<td>loss</td>
</tr>
<tr>
<td>2</td>
<td>no weigh in</td>
</tr>
<tr>
<td>3</td>
<td>no weigh in</td>
</tr>
<tr>
<td>4</td>
<td>gain</td>
</tr>
<tr>
<td>5</td>
<td>same</td>
</tr>
</tbody>
</table>
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:

``` SAS
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 7.27

<table>
<thead>
<tr>
<th>OBS</th>
<th>ID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1011</td>
<td>376</td>
</tr>
<tr>
<td>2</td>
<td>1026</td>
<td>269    &lt;=== value remains 269</td>
</tr>
<tr>
<td>3</td>
<td>1028</td>
<td>374</td>
</tr>
<tr>
<td>4</td>
<td>1034</td>
<td>A      &lt;=== special missing value</td>
</tr>
<tr>
<td>5</td>
<td>1057</td>
<td>.      &lt;=== regular missing value</td>
</tr>
</tbody>
</table>

See Also

Statements:
WHERE Statement

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. See “Understanding SAS Indexes” in the “SAS Data Files” section 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 the POINT= option in the SET and MODIFY statements.

You can apply OBS= and FIRSTOBS= processing to WHERE processing. For more information, see “Processing a Segment of Data That is Conditionally Selected” in the “WHERE-Expression Processing” section of SAS Language Reference: Concepts.

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 DATALINES statement.

For each iteration of the DATA step, the first operation SAS 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 Programs” 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 in order to subset the original data set for processing by the procedure. The Base 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:

```plaintext
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 7.11 on page 1423. For the rules SAS follows when it evaluates WHERE expressions, see “WHERE-Expression Processing” in *SAS Language Reference: Concepts*.

**Table 7.11 WHERE Statement Operators**

<table>
<thead>
<tr>
<th>Operator Type</th>
<th>Symbol or Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>*</td>
<td>multiplication</td>
</tr>
<tr>
<td></td>
<td>/</td>
<td>division</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>addition</td>
</tr>
<tr>
<td></td>
<td>–</td>
<td>subtraction</td>
</tr>
<tr>
<td></td>
<td>**</td>
<td>exponentiation</td>
</tr>
</tbody>
</table>

Comparison 4
### Operator Type

<table>
<thead>
<tr>
<th>Symbol or Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>= or EQ</td>
<td>equal to</td>
</tr>
<tr>
<td>^=, -=, -=, or NE</td>
<td>not equal to</td>
</tr>
<tr>
<td>&gt; or GT</td>
<td>greater than</td>
</tr>
<tr>
<td>&lt; or LT</td>
<td>less than</td>
</tr>
<tr>
<td>&gt;= or GE</td>
<td>greater than or equal to</td>
</tr>
<tr>
<td>&lt;= or LE</td>
<td>less than or equal to</td>
</tr>
<tr>
<td>IN</td>
<td>equal to one of a list</td>
</tr>
</tbody>
</table>

### Logical (Boolean)

<table>
<thead>
<tr>
<th>Symbol or Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp; or AND</td>
<td>logical and</td>
</tr>
<tr>
<td></td>
<td>or OR</td>
</tr>
<tr>
<td>~, ^, ~ or NOT</td>
<td>logical not</td>
</tr>
</tbody>
</table>

### Other

<table>
<thead>
<tr>
<th>Symbol or Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>()</td>
<td>indicate order of evaluation</td>
</tr>
<tr>
<td>+ prefix</td>
<td>positive number</td>
</tr>
<tr>
<td>− prefix</td>
<td>negative number</td>
</tr>
</tbody>
</table>

### WHERE Expression Only

<table>
<thead>
<tr>
<th>Symbol or Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BETWEEN–AND</td>
<td>an inclusive range</td>
</tr>
<tr>
<td>? or CONTAINS</td>
<td>a character string</td>
</tr>
<tr>
<td>IS NULL or IS MISSING</td>
<td>missing values</td>
</tr>
<tr>
<td>LIKE</td>
<td>match patterns</td>
</tr>
<tr>
<td>^=*</td>
<td>sounds-like</td>
</tr>
<tr>
<td>SAME-AND</td>
<td>add clauses to an existing WHERE statement without retyping original one</td>
</tr>
</tbody>
</table>

---

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. Two OR symbols ( | | ), two broken vertical bars ( | | ), or two exclamation points (!!) indicate concatenation. Use the character available on your keyboard.
4. You can use the colon modifier (:) with any of the comparison operators in order to compare only a specified prefix of a character string.

### 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.

```sas
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:
  ```sas
  where empnum between 500 and 1000;
  ```

- Using CONTAINS:
  ```sas
  where company ? 'bay';
  where company contains 'bay';
  ```

- Using IS NULL and IS MISSING:
  ```sas
  where name is null;
  where name is missing;
  ```

- Using LIKE to select all names that start with the letter D:
  ```sas
  where name like 'D%';
  ```

- Using LIKE to match patterns from a list of the following names:
  ```sas
  Diana
  Diane
  ```
Dianna
Dianthus
Dyan

<table>
<thead>
<tr>
<th>WHERE Statement</th>
<th>Name Selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>where name like ‘D_an’;</td>
<td>Dyan</td>
</tr>
<tr>
<td>where name like ‘D_an_’;</td>
<td>Diana, Diane</td>
</tr>
<tr>
<td>where name like ‘D_an__’;</td>
<td>Dianna</td>
</tr>
<tr>
<td>where name like ‘D_an%’;</td>
<td>all names from list</td>
</tr>
</tbody>
</table>

- 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= Data Set Option” on page 61
Statement:

- “IF Statement, Subsetting” on page 1214

SAS SQL Query Window User’s Guide
SAS/IML User’s Guide
Base SAS Procedures Guide

“SAS Indexes” in SAS Language Reference: Concepts
“WHERE-Expression Processing” in SAS Language Reference: Concepts
“BY-Group Processing” in SAS Language Reference: Concepts

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

\[ \text{libref.catalog.keys-entry} \]

**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

\[ \text{libref.catalog.pmenu-entry} \]

**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 1429.

**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 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 Statement” on page 1141.

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).

Tip: Although you can specify either row or column first, the examples in this documentation 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

* You cannot use the WINDOW statement in batch mode because no terminal is connected to a batch executing process.
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 \textit{n} within the window.
  \textbf{Range:} \textit{n} must be a positive integer.

\#\textit{numeric-variable}
  specifies the row within the window that is given by the value of
  \textit{numeric-variable}.
  \textbf{Restriction:} \#\textit{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.

\#(\textit{expression})
  specifies the row within the window that is given by the value of \textit{expression}.
  \textbf{Restriction:} \textit{expression} can contain array references and must evaluate to a
  positive integer.
  \textbf{Restriction:} Enclose \textit{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 \textit{n} within the window.
  \textbf{Restriction:} \textit{n} must be a positive integer.

@\textit{numeric-variable}
  specifies the column within the window that is given by the value of
  \textit{numeric-variable}.
  \textbf{Restriction:} \textit{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.

@(\textit{expression})
  specifies the column within the window that is given by the value of
  \textit{expression}.
  \textbf{Restriction:} \textit{expression} can contain array references and must evaluate to a
  positive integer.
  \textbf{Restriction:} Enclose \textit{expression} in parentheses.

+n
  moves the pointer \textit{n} columns.
  \textbf{Range:} \textit{n} must be a positive integer.

+n\textit{numeric-variable}
  moves the pointer the number of columns that is given by the
  \textit{numeric-variable}.
  \textbf{Restriction:} +\textit{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.

\textit{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 1436

**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 1436

**_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 1436

**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:

```sas
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 Statement” on page 1141
- “The PMENU Procedure” in *Base SAS Procedures Guide*

---

**X Statement**

**Issues an operating-environment command from within a SAS session**

**Valid:** anywhere

**Category:** Operating Environment

**See:** X Statement in the documentation for your 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 Macro Language: Reference* 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 Routine” on page 403

Function:

“SYSTEM Function” on page 853
CHAPTER 8

SAS System Options

Definition of System Options 1443
Syntax 1443
Specifying System Options in an OPTIONS Statement 1443
Specifying Hexadecimal Values 1443
Using SAS System Options 1443
  Default Settings 1443
  Determining Which Settings Are in Effect 1444
  Determining Which SAS System Options Are Restricted 1444
  Determining How a SAS System Option Value Was Set 1445
  Obtaining Descriptive Information about a System Option 1445
  Changing SAS System Option Settings 1446
  How Long System Option Settings Are in Effect 1447
  Order of Precedence 1447
  Interaction with Data Set Options 1448
Comparisons 1448
SAS System Options by Category 1449
Dictionary 1458
  APPLETLOC= System Option 1458
  ARMAGENT= System Option 1459
  ARMLOC= System Option 1460
  ARMSUBSYS= System Option 1461
  ASYNCHIO System Option 1477
  AUTHPROVIDERDOMAIN= System Option 1477
  AUTOSAVELOC= System Option 1479
  BATCH System Option 1479
  BINDING= System Option 1480
  BOTTOMMARGIN= System Option 1481
  BUFNO= System Option 1482
  BUFSIZE= System Option 1483
  BYERR System Option 1485
  BYLINE System Option 1486
  BYSORTED System Option 1487
  CAPS System Option 1488
  CARDIMAGE System Option 1489
  CATCACHE= System Option 1490
  CBUFNO= System Option 1491
  CENTER System Option 1492
  CHARCODE System Option 1492
  CLEANUP System Option 1493
  CMDDMAC System Option 1495
  CMPLIB= System Option 1496
CMPOPT= System Option 1497
COLLATE System Option 1499
COLORPRINTING System Option 1500
COMPRESS= System Option 1500
COPIES= System Option 1502
CPUCOUNT= System Option 1503
CPUID System Option 1504
DATASTMTCHK= System Option 1505
DATE System Option 1506
DATESTYLE= System Option 1506
DETAILS System Option 1507
DEVICE= System Option 1508
DFLANG= System Option 1509
DKRICOND= System Option 1509
DKROCOND= System Option 1510
DLDMGACTION= System Option 1511
DMR System Option 1511
DMS System Option 1512
DMSEXP System Option 1513
DMSLOGSIZE= System Option 1514
DMSOUTSIZE= System Option 1515
DMSSYNCHK System Option 1516
DSNFERR System Option 1517
DTRESET System Option 1518
DUPLEX System Option 1519
ECHOAUTO System Option 1519
EMAILAUTHPROTOCOL= System Option 1520
EMAILHOST System Option 1521
EMAILID= System Option 1522
EMAILPORT System Option 1523
EMAILPW= System Option 1524
ENGINE= System Option 1525
ERRORABEND System Option 1525
ERRORBYABEND System Option 1526
ERRORCHECK= System Option 1527
ERRORS= System Option 1527
EXPLORER System Option 1528
FIRSTOBS= System Option 1529
FMTERR System Option 1531
FMTSEARCH= System Option 1532
FONTSLOC= System Option 1533
FORMCHAR= System Option 1533
FORMDLIM= System Option 1534
FORMS= System Option 1535
GISMAPS= System Option 1536
GWINDOW System Option 1536
HELPENCMD System Option 1537
IBUFSIZE= System Option 1538
IMPLMAC System Option 1539
INITCMD System Option 1539
INITSTMT= System Option 1541
INVALIDDATA= System Option 1542
LABEL System Option 1543
_LAST_= System Option 1543
LEFTMARGIN= System Option 1544
LINESIZE= System Option 1545
LOGPARM= System Option 1546
MACRO System Option 1550
MAPS= System Option 1550
MAUTOLOCDISPLAY System Option 1550
MAUTOSOURCE System Option 1551
MCOMPILENOTE= System Option 1551
MERGENOBY System Option 1551
MERROR System Option 1552
META System Option 1552
META System Option 1552
META System Option 1552
META System Option 1552
META System Option 1552
META System Option 1552
MFILE System Option 1565
MINDELIMITER= System Option 1565
MISSING= System Option 1565
MLOGIC System Option 1566
MLOGICNEST System Option 1566
MPRINT System Option 1567
MPRINTNEST System Option 1567
MRECALL System Option 1567
MSYMTABMAX= System Option 1570
MLOGIC System Option 1566
NOTES System Option 1570
NUMBER System Option 1571
OBJECTSERVER System Option 1571
OBS= System Option 1572
ORIENTATION= System Option 1578
OVP System Option 1579
PAGEBREAKINITIAL System Option 1579
PAGENO= System Option 1580
PAGESIZE= System Option 1581
PAPERDEST= System Option 1581
PAPERSIZE= System Option 1582
PAPERSOURCE= System Option 1583
PAPERTYPE= System Option 1584
PARAM= System Option 1585
PARMCARDS= System Option 1586
PRINTERPATH= System Option 1586
PRINTINIT System Option 1587
PRINTMSGLIST System Option 1588
Chapter 8

QUOTELENMAX System Option 1589
REPLACE System Option 1590
REUSE= System Option 1591
RIGHTMARGIN= System Option 1592
RSASUSER System Option 1593
S= System Option 1594
S2= System Option 1596
SASAUTOS= System Option 1596
SASHELP= System Option 1597
SASMSTORE= System Option 1597
SASUSER= System Option 1598
SEQ= System Option 1598
SERROR System Option 1599
SETINIT System Option 1599
SKIP= System Option 1600
SOLUTIONS System Option 1601
SORTDUP= System Option 1601
SORTEQUALS System Option 1602
SORTSEQ= System Option 1603
SORTSIZE= System Option 1603
SOURCE System Option 1604
SOURCE2 System Option 1605
SPOOL System Option 1606
STARTLIB System Option 1607
SUMSIZE= System Option 1607
SYMBOLGEN System Option 1608
SYNTAXCHECK System Option 1609
SYSPARM= System Option 1610
SYSPRINTFONT= System Option 1610
TERMINAL System Option 1612
TERMSTMT= System Option 1613
TEXTURELOC= System Option 1614
THREADS System Option 1615
TOOLSMENU System Option 1616
TOPMARGIN= System Option 1617
TRAINLOC= System Option 1618
TRANTAB= System Option 1618
UNIVERSALPRINT System Option 1619
USER= System Option 1619
UTILLOC= System Option 1620
UUIDCOUNT= System Option 1621
UUIDGENDHOST= System Option 1622
V6CREATEUPDATE= System Option 1623
VALIDFMTNAME= System Option 1624
VALIDVARNAME= System Option 1625
VIEWMENU System Option 1627
VNFERR System Option 1627
WORK= System Option 1628
WORKINIT System Option 1629
WORKTERM System Option 1630
YEARCUTOFF= System Option 1630
Definition of System Options

System options are instructions that affect your SAS session. They control the way that SAS performs operations such as SAS System initialization, hardware and software interfacing, and the input, processing, and output of jobs and SAS files.

Syntax

Specifying System Options in an OPTIONS Statement

The syntax for specifying system options in an OPTIONS statement is

```
OPTIONS option(s);
```

where

- `option` specifies one or more SAS system options that you want to change.

The following example shows how to use the system options NODATE and LINESIZE= in an OPTIONS statement:

```
options nodate linesize=72;
```

Operating Environment Information: On the command line or in a configuration file, the syntax is specific to your operating environment. For details, see the SAS documentation for your operating environment.

Specifying Hexadecimal Values

Hexadecimal values for system options must begin with a number (0-9), followed by an X. For example, the following OPTIONS statement sets the linesize to 160 using a hexadecimal number:

```
options linesize=0a0x;
```

Using SAS System Options

Default Settings

Operating Environment Information: SAS system options are initialized with default settings when SAS is invoked. However, the default settings for some SAS system options vary both by operating environment and by site. For details, see the SAS documentation for your operating environment.
Determining Which Settings Are in Effect

To determine which settings are in effect for SAS system options, use one of the following:

OPLIST system option
  Writes to the SAS log the system options that were specified on the SAS invocation command line. (See the SAS documentation for your operating environment for more information.)

VERBOSE system option
  Writes to the SAS log the system options that were specified in the configuration file and on the SAS invocation command line.

SAS System Options window
  Lists all system option settings.

OPTIONS procedure
  Writes system option settings to the SAS log. To display the settings of system options with a specific functionality, such as error handling, use the GROUP= option:
  
  proc options GROUP=errorhandling;
  run;

  (See the *Base SAS Procedures Guide* for more information.)

GETOPTION function
  Returns the value of a specified system option.

VOPTION Dictionary table
  Located in the SASHELP library, VOPTION contains a list of all current system option settings. You can view this table with SAS Explorer, or you can extract information from the VOPTION table using PROC SQL.

dictionary.options SQL table
  Accessed with the SQL procedure, this table lists the system options that are in effect.

Determining Which SAS System Options Are Restricted

To determine which system options are restricted by your system administrator, use the RESTRICT option of the OPTIONS procedure. The RESTRICT option displays the option’s value, scope, and setting. In the following example, the SAS log shows that only one option, CMPOPT, is restricted:

```
proc options restrict;
run;
```

Output 8.1  Restricted Option Information

```
 1 proc options restrict;
 2 run;
SAS (r) Proprietary Software Release 9.1  TS1B0

Option Value Information For SAS Option CMPOPT
  Option Value: (NOEXTRAMATH NOMISSCHECK NOPRECISE NOGUARDCHECK)
  Option Scope: SAS Session
  How option value set:  Site Administrator Restricted
```
The OPTIONS procedure displays this information for all options that are restricted. If your site administrator has not restricted any options, then the following message appears in the SAS log:

Your site administrator has not restricted any options.

### Determining How a SAS System Option Value Was Set

To determine how a system option value was set, use the OPTIONS procedure with the VALUE option specified in the OPTIONS statement. The VALUE option displays the specified option’s value and scope. For example, the following statements write a message to the SAS log that tells you how the option value for the system option CENTER was set:

```sas
proc options option=center value;
run;
```

The following partial SAS log shows that the option value for CENTER was the shipped default.

**Output 8.2  Option Value Information for the System Option CENTER**

```
  2 proc options option=center value;
  3 run;
```

Option Value Information for SAS Option CENTER

<table>
<thead>
<tr>
<th>Option Value: CENTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Option Scope: Default</td>
</tr>
<tr>
<td>How option value set: Shipped Default</td>
</tr>
</tbody>
</table>

If no value is assigned to a character system option, then SAS assigns the option a value of ‘ ’ (a space between two single quotation marks) and *Option Value* displays a blank space.

### Obtaining Descriptive Information about a System Option

You can quickly obtain basic descriptive information about a system option by specifying the DEFINE option in the PROC OPTIONS statement.

The DEFINE option writes the following descriptive information about a system option to the SAS log:

- description
- type
- when in the SAS session it can be set
- if it can be restricted by the system administrator
- if the OPTSAVE procedure or the DMOPTSAVE command will save the option.

For example, the following statements write a message to the SAS log that contains descriptive information about the system option CENTER:

```sas
proc options option=center define;
run;
```
Output 8.3  Descriptive Information for the System Option CENTER

This partial SAS log tells you specific information about the system option CENTER.

```
1  proc options option=center define;
2   run;
CENTER
Option Definition Information for SAS Option CENTER
   Group= LISTCONTROL
   Group Description: Procedure output and display settings
   Description: Center SAS procedure output
   Type: The option value is of type BOOLEAN
   When Can Set: Startup or anytime during the SAS Session
   Restricted: Your Site Administrator can restrict modification of this option
   Optsave: Proc Optsave or command Dmoptsave will save this option.
```

### Changing SAS System Option Settings

SAS provides default settings for SAS system options. You can override the default settings of any unrestricted system option in several ways. Depending on the function of the system options, you can specify a setting

- **on the command line or in a configuration file:** You can specify any unrestricted SAS system option setting either on the SAS command line or in a configuration file. If you use the same option settings frequently, then it is usually more convenient to specify the options in a configuration file, rather than on the command line. Either method sets your SAS system options during SAS invocation. Many SAS system option settings can be specified only during SAS invocation. Descriptions of the individual options provide details.

- **in an OPTIONS statement:** You can specify an OPTIONS statement at any time during a session except within data lines or parmcard lines. Settings remain in effect throughout the current program or process unless you reset them with another OPTIONS statement or change them in the SAS System Options window. You can also place an OPTIONS statement in an autoexec file.

- **in a SAS System Options window:** If you are in a windowing environment, type `options` in the toolbox or on the command line to open the SAS System Options window. The SAS System Options window lists the names of the SAS system option groups. You can then expand the groups to see the option names and to change their current settings. Alternatively, you can use the Find Option command in the Options pop-up menu to go directly to an option. Changes take effect immediately and remain in effect throughout the session unless you reset them with an OPTIONS statement or change them in the SAS System Options window.

**Operating Environment Information:** Under UNIX, Open VMS, and z/OS operating environments, SAS system options can be restricted by a site administrator so that after they are set by the administrator, they cannot be changed by a user. Depending upon your operating environment, system options can be restricted globally, by group, or by user. You can use the OPTIONS procedure to determine which options are restricted. For more information about how to restrict options, see the SAS configuration guide for your operating environment. For more information about the OPTIONS procedure, see the SAS documentation for your operating environment.
How Long System Option Settings Are in Effect

When you specify a SAS system option setting, the setting applies to the next step and to all subsequent steps for the duration of the SAS session, or until you reset, as shown:

data one;
   set items;
run;

   /* option applies to all subsequent steps */
   options obs=5;

   /* printing ends with the fifth observation */
   proc print data=one;
   run;

   /* the SET statement stops reading after the fifth observation */
   data two;
   set items;
   run;

To read more than five observations, you must reset the OBS= system option. For more information, see “OBS= System Option” on page 1572.

Order of Precedence

If the same system option appears in more than one place, the order of precedence from highest to lowest is

1 OPTIONS statement and SAS System Options window
2 autoexec file (that contains an OPTIONS statement)
3 command-line specification
4 configuration file specification
5 SAS system default settings.

Operating Environment Information:  In some operating environments, you can specify system options in other places. See the SAS documentation for your operating environment.

The following table shows the order of precedence that SAS uses for execution mode options. These options are a subset of the SAS invocation options and are specified on the command line during SAS invocation.

Table 8.1  Order of Precedence for SAS Execution Mode Options

<table>
<thead>
<tr>
<th>Execution Mode Option</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBJECTSERVER</td>
<td>Highest</td>
</tr>
<tr>
<td>DMR</td>
<td>2nd</td>
</tr>
<tr>
<td>INITCMD</td>
<td>3rd</td>
</tr>
</tbody>
</table>
The order of precedence of SAS execution mode options consists of the following rules:

- SAS uses the execution mode option with the highest precedence.
- If you specify more than one execution mode option of equal precedence, SAS uses only the last option listed.

See the descriptions of the individual options for more details.

### Interaction with Data Set Options

Many system options and data set options share the same name and have the same function. System options remain in effect for all DATA and PROC steps in a SAS job or session until their settings are changed. A data set option, however, overrides a system option only for the particular data set in the step in which it appears.

In this example, the OBS= system option in the OPTIONS statement specifies that only the first 100 observations will be read from any data set within the SAS job. The OBS= data set option in the SET statement, however, overrides the system option and specifies that only the first 5 observations will be read from data set TWO. The PROC PRINT step uses the system option setting and reads and prints the first 100 observations from data set THREE:

```sas
options obs=100;

data one;
  set two(obs=5);
run;

proc print data=three;
run;
```

### Comparisons

Note the differences between system options, data set options, and statement options.

- **system options** remain in effect for all DATA and PROC steps in a SAS job or current process unless they are respecified.

- **data set options** apply to the processing of the SAS data set with which they appear. Some data set options have corresponding system options or LIBNAME statement options. For an individual data set, you can use the data set option to override the setting of these other options.

- **statement options** control the action of the statement in which they appear. Options in global statements, such as in the LIBNAME statement, can have a broader impact.
### Table 8.2  Categories and Descriptions of SAS System Options

<table>
<thead>
<tr>
<th>Category</th>
<th>SAS System Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communications: Email</td>
<td>“EMAILAUTHPROTOCOL= System Option” on page 1520</td>
<td>Specifies the authentication protocol for SMTP E-mail</td>
</tr>
<tr>
<td></td>
<td>“EMAILHOST System Option” on page 1521</td>
<td>Specifies the Simple Mail Transfer Protocol (SMTP) server that supports e-mail access for your site</td>
</tr>
<tr>
<td></td>
<td>“EMAILID= System Option” on page 1522</td>
<td>Specifies the identity of the individual sending e-mail from within SAS</td>
</tr>
<tr>
<td></td>
<td>“EMAILPORT System Option” on page 1523</td>
<td>Specifies the port to which the Simple Mail Transfer Protocol (SMTP) server is attached</td>
</tr>
<tr>
<td></td>
<td>“EMAILPW= System Option” on page 1524</td>
<td>Specifies your e-mail login password</td>
</tr>
<tr>
<td>Communications: Meta Data</td>
<td>“METAUTORESOURCES= System Option” on page 1552</td>
<td>Identifies what resources to be assigned at SAS startup</td>
</tr>
<tr>
<td></td>
<td>“METACONNECT= System Option” on page 1553</td>
<td>Identifies the named connection from the metadata user profiles to use as the default values for logging in to the SAS Metadata Server</td>
</tr>
<tr>
<td></td>
<td>“METAENCRYPTALG= System Option” on page 1555</td>
<td>Specifies the type of encryption to use when communicating with a SAS Metadata Server</td>
</tr>
<tr>
<td></td>
<td>“METAENCRYPTLEVEL= System Option” on page 1556</td>
<td>Specifies what is to be encrypted when communicating with a SAS Metadata Server</td>
</tr>
<tr>
<td></td>
<td>“METAID= System Option” on page 1557</td>
<td>Identifies the current installed version of SAS on the SAS Metadata Server</td>
</tr>
<tr>
<td></td>
<td>“METAPASS= System Option” on page 1557</td>
<td>Sets the default password for the SAS Metadata Server</td>
</tr>
<tr>
<td></td>
<td>“METAPORT= System Option” on page 1559</td>
<td>Sets the TCP port for the SAS Metadata Server</td>
</tr>
<tr>
<td></td>
<td>“METAPROFILE= System Option” on page 1559</td>
<td>Identifies the file that contains SAS Metadata Server user profiles</td>
</tr>
<tr>
<td></td>
<td>“METAPROTOCOL= System Option” on page 1561</td>
<td>Sets the network protocol for communicating with the SAS Metadata Server</td>
</tr>
<tr>
<td></td>
<td>“METAREPOSITORY= System Option” on page 1562</td>
<td>Sets the default SAS Metadata Repository to use on the SAS Metadata Server</td>
</tr>
<tr>
<td></td>
<td>“METASERVER= System Option” on page 1563</td>
<td>Sets the address of the SAS Metadata Server</td>
</tr>
<tr>
<td>Category</td>
<td>SAS System Options</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Environment control:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Display</td>
<td>“METAUSER= System Option” on page 1564</td>
<td>Sets the default user identification for logging onto the SAS Metadata Server</td>
</tr>
<tr>
<td></td>
<td>“AUTOSAVELOC= System Option” on page 1479</td>
<td>Specifies the location of the Program Editor autosave file</td>
</tr>
<tr>
<td></td>
<td>“CHARCODE System Option” on page 1492</td>
<td>Determines whether character combinations are substituted for special characters that are not on the keyboard</td>
</tr>
<tr>
<td></td>
<td>“DMSLOGSIZE= System Option” on page 1514</td>
<td>Specifies the maximum number of rows that the SAS windowing environment Log window can display</td>
</tr>
<tr>
<td></td>
<td>“DMSOUTSIZE= System Option” on page 1515</td>
<td>Specifies the maximum number of rows that the SAS windowing environment Output window can display</td>
</tr>
<tr>
<td></td>
<td>“FONTSLOC= System Option” on page 1533</td>
<td>Specifies the location that contains the SAS fonts that are loaded by some Universal Printer drivers</td>
</tr>
<tr>
<td></td>
<td>“FORMS= System Option” on page 1535</td>
<td>Specifies the default form that is used for windowing output</td>
</tr>
<tr>
<td></td>
<td>“SOLUTIONS System Option” on page 1601</td>
<td>Specifies whether the SOLUTIONS menu choice appears in all SAS windows and whether the SOLUTIONS folder appears in the SAS Explorer window</td>
</tr>
<tr>
<td></td>
<td>“TOOLSMENU System Option” on page 1616</td>
<td>Specifies to include or suppress the Tools menu in windows that display menus</td>
</tr>
<tr>
<td></td>
<td>“VIEWMENU System Option” on page 1627</td>
<td>Specifies to include or suppress the View menu in windows that display menus</td>
</tr>
<tr>
<td>Error handling</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>“BYERR System Option” on page 1485</td>
<td>Controls whether SAS generates an error message and sets the error flag when a <em>NULL</em> data set is used in the SORT procedure</td>
</tr>
<tr>
<td></td>
<td>“CLEANUP System Option” on page 1493</td>
<td>Specifies how to handle an out-of-resource condition</td>
</tr>
<tr>
<td></td>
<td>“DMSSYNCHK System Option” on page 1516</td>
<td>Enables syntax check mode for multiple steps in the SAS windowing environment</td>
</tr>
<tr>
<td></td>
<td>“DSNFERR System Option” on page 1517</td>
<td>Controls how SAS responds when a SAS data set is not found</td>
</tr>
<tr>
<td></td>
<td>“ERRORABEND System Option” on page 1525</td>
<td>Specifies how SAS responds to errors</td>
</tr>
<tr>
<td></td>
<td>“ERRORBYABEND System Option” on page 1526</td>
<td>Specifies how SAS responds to BY-group error conditions</td>
</tr>
<tr>
<td></td>
<td>“ERRORCHECK= System Option” on page 1527</td>
<td>Controls error handling</td>
</tr>
<tr>
<td></td>
<td>“ERRORS= System Option” on page 1527</td>
<td>Controls the maximum number of observations for which complete error messages are printed</td>
</tr>
<tr>
<td></td>
<td>“FMTERR System Option” on page 1531</td>
<td>Determines whether SAS generates an error message when a format of a variable cannot be found</td>
</tr>
<tr>
<td></td>
<td>“QUOTELENMAX System Option” on page 1589</td>
<td>Specifies that SAS write to the SAS log a warning about the maximum length for strings in quotation marks</td>
</tr>
<tr>
<td>Category</td>
<td>SAS System Options</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>----------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>“SYNTAXCHECK System Option” on page 1609</td>
<td>Enables syntax check mode for multiple steps in non-interactive or batch SAS sessions</td>
<td></td>
</tr>
<tr>
<td>“VNFERR System Option” on page 1627</td>
<td>Controls how SAS responds when a <em>NULL</em> data set is used</td>
<td></td>
</tr>
<tr>
<td>Environment control: Files</td>
<td>“APPLETLOC= System Option” on page 1458</td>
<td>Specifies the location of Java applets</td>
</tr>
<tr>
<td></td>
<td>“FMTSEARCH= System Option” on page 1532</td>
<td>Controls the order in which format catalogs are searched</td>
</tr>
<tr>
<td></td>
<td>“HELPENCMD System Option” on page 1537</td>
<td>Controls whether SAS uses the English version or the translated version of the keyword list for the command–line Help</td>
</tr>
<tr>
<td></td>
<td>“NEWS= System Option” on page 1570</td>
<td>Specifies a file that contains messages to be written to the SAS log</td>
</tr>
<tr>
<td></td>
<td>“PARM= System Option” on page 1585</td>
<td>Specifies a parameter string that is passed to an external program</td>
</tr>
<tr>
<td></td>
<td>“PARMCARDS= System Option” on page 1586</td>
<td>Specifies the file reference to use as the PARMCARDS file</td>
</tr>
<tr>
<td></td>
<td>“RSASUSER System Option” on page 1593</td>
<td>Controls access to the SASUSER library</td>
</tr>
<tr>
<td></td>
<td>“SASAUTOS= System Option” on page 1596</td>
<td>Specifies the autocall macro library</td>
</tr>
<tr>
<td></td>
<td>“SASHELP= System Option” on page 1597</td>
<td>Specifies the location of the SASHELP library</td>
</tr>
<tr>
<td></td>
<td>“SASUSER= System Option” on page 1598</td>
<td>Specifies the name of the SASUSER library</td>
</tr>
<tr>
<td></td>
<td>“SYSPARM= System Option” on page 1610</td>
<td>Specifies a character string that can be passed to SAS programs</td>
</tr>
<tr>
<td></td>
<td>“TRAINLOC= System Option” on page 1618</td>
<td>Specifies the base location of SAS online training courses</td>
</tr>
<tr>
<td></td>
<td>“USER= System Option” on page 1619</td>
<td>Specifies the default permanent SAS data library</td>
</tr>
<tr>
<td></td>
<td>“UUIDCOUNT= System Option” on page 1621</td>
<td>Specifies the number of UUIDs to acquire each time the UUID Generator Daemon is used</td>
</tr>
<tr>
<td></td>
<td>“UUIDGENDHOST= System Option” on page 1622</td>
<td>Identifies the host and the port of the UUID Generator Daemon</td>
</tr>
<tr>
<td></td>
<td>“V6CREATEUPDATE= System Option” on page 1623</td>
<td>Controls or monitors the creation of new Version 6 data sets or the updating of existing Version 6 data sets.</td>
</tr>
<tr>
<td></td>
<td>“WORK= System Option” on page 1628</td>
<td>Specifies the WORK data library</td>
</tr>
<tr>
<td></td>
<td>“WORKINIT System Option” on page 1629</td>
<td>Initializes the WORK data library</td>
</tr>
<tr>
<td>Category</td>
<td>SAS System Options</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Environment control: Initialization and operation</td>
<td>“WORKTERM System Option” on page 1630</td>
<td>Controls whether SAS erases WORK files at the termination of a SAS session</td>
</tr>
<tr>
<td></td>
<td>“AUTHPROVIDERDOMAIN= System Option” on page 1477</td>
<td>Associates a domain suffix with an authentication provider</td>
</tr>
<tr>
<td></td>
<td>“BATCH System Option” on page 1479</td>
<td>Specifies whether batch settings for LINESIZE, OVP, PAGESIZE, and SOURCE are in effect when SAS executes</td>
</tr>
<tr>
<td></td>
<td>“DMR System Option” on page 1511</td>
<td>Controls the ability to invoke a remote SAS session so that you can run SAS/CONNECT software</td>
</tr>
<tr>
<td></td>
<td>“DMS System Option” on page 1512</td>
<td>Invokes the SAS windowing environment</td>
</tr>
<tr>
<td></td>
<td>“DMSEXPF System Option” on page 1513</td>
<td>Invokes SAS with the Explorer, Program Editor, Log, Output, and Results windows</td>
</tr>
<tr>
<td></td>
<td>“EXPLORER System Option” on page 1528</td>
<td>Controls whether you invoke SAS with only the Explorer window</td>
</tr>
<tr>
<td></td>
<td>“INITCMD System Option” on page 1539</td>
<td>Suppresses the Log, Output, and Program Editor windows when you enter a SAS/AF application</td>
</tr>
<tr>
<td></td>
<td>“INITSTMT= System Option” on page 1541</td>
<td>Specifies a SAS statement to be executed after any statements in the autoexec file and before any statements from the SYSIN= file</td>
</tr>
<tr>
<td></td>
<td>“MULTENVAPPL System Option” on page 1569</td>
<td>Controls whether SAS/AF, SAS/FSP, and Base SAS windowing applications use a default on an operating environment specific font selector window</td>
</tr>
<tr>
<td></td>
<td>“OBJECTSERVER System Option” on page 1571</td>
<td>Specifies whether to put the SAS session into DCOM/ CORBA server mode</td>
</tr>
<tr>
<td></td>
<td>“TERMINAL System Option” on page 1612</td>
<td>Determines whether SAS evaluates the execution mode and, if needed, resets the option</td>
</tr>
<tr>
<td></td>
<td>“TERMSTMT= System Option” on page 1613</td>
<td>Specifies the SAS statements to be executed when the SAS session is terminated</td>
</tr>
<tr>
<td>Environment control: Language control</td>
<td>“DATESTYLE= System Option” on page 1506</td>
<td>Identifies the sequence of month, day, and year when ANYTDTDE, ANYDTSDTM, or ANYDTTME informat data is ambiguous</td>
</tr>
<tr>
<td></td>
<td>“DFLANG= System Option” on page 1509</td>
<td>Specifies language for international date informs and formats</td>
</tr>
<tr>
<td></td>
<td>“PAPERSIZE= System Option” on page 1582</td>
<td>Specifies to a printer the paper size to use</td>
</tr>
<tr>
<td></td>
<td>“TRANTAB= System Option” on page 1618</td>
<td>Specifies the translation tables that are used by various parts of SAS</td>
</tr>
<tr>
<td>Files: External files</td>
<td>“STARTLIB System Option” on page 1607</td>
<td>Allows previous library references (librefs) to persist in a new SAS session</td>
</tr>
<tr>
<td>Files: SAS Files</td>
<td>“ASYNCHIO System Option” on page 1477</td>
<td>Specifies whether asynchronous I/O is enabled</td>
</tr>
<tr>
<td>Category</td>
<td>SAS System Options</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>&quot;BUFNO= System Option&quot; on page 1482</td>
<td>Specifies the number of buffers to be allocated for processing SAS data sets</td>
<td></td>
</tr>
<tr>
<td>&quot;BUFSIZE= System Option&quot; on page 1483</td>
<td>Specifies the permanent buffer page size for output SAS data sets</td>
<td></td>
</tr>
<tr>
<td>&quot;CATCACHE= System Option&quot; on page 1490</td>
<td>Specifies the number of SAS catalogs to keep open</td>
<td></td>
</tr>
<tr>
<td>&quot;CBUFNO= System Option&quot; on page 1491</td>
<td>Controls the number of extra page buffers to allocate for each open SAS catalog</td>
<td></td>
</tr>
<tr>
<td>&quot;CMPLIB= System Option&quot; on page 1496</td>
<td>Specifies one or more SAS catalogs containing compiler subroutines to include during program compilation</td>
<td></td>
</tr>
<tr>
<td>&quot;COMPRESS= System Option&quot; on page 1500</td>
<td>Controls the compression of observations in output SAS data sets</td>
<td></td>
</tr>
<tr>
<td>&quot;DATASTMTCHK= System Option&quot; on page 1505</td>
<td>Prevents certain errors by controlling the SAS keywords that are allowed in the DATA statement</td>
<td></td>
</tr>
<tr>
<td>&quot;DKRICOND= System Option&quot; on page 1509</td>
<td>Controls the level of error detection for input data sets during processing of DROP=, KEEP=, and RENAME= data set options</td>
<td></td>
</tr>
<tr>
<td>&quot;DKROCOND= System Option&quot; on page 1510</td>
<td>Controls the level of error detection for output data sets during the processing of DROP=, KEEP=, and RENAME= data set options and the corresponding DATA step statements</td>
<td></td>
</tr>
<tr>
<td>&quot;DLDMGACTION= System Option&quot; on page 1511</td>
<td>Specifies what type of action to take when a SAS catalog or a SAS data set in a SAS data library is detected as damaged</td>
<td></td>
</tr>
<tr>
<td>&quot;ENGINE= System Option&quot; on page 1525</td>
<td>Specifies the default access method for SAS libraries</td>
<td></td>
</tr>
<tr>
<td>&quot;FIRSTOBS= System Option&quot; on page 1529</td>
<td>Specifies which observation or record SAS processes first</td>
<td></td>
</tr>
<tr>
<td>&quot;IBUFSIZE= System Option&quot; on page 1538</td>
<td>Specifies the buffer page size for an index file</td>
<td></td>
</tr>
<tr>
<td>&quot;<em>LAST</em>= System Option&quot; on page 1543</td>
<td>Specifies the most recently created data set</td>
<td></td>
</tr>
<tr>
<td>&quot;MERGENOBY System Option&quot; on page 1551</td>
<td>Controls whether a message is issued when MERGE processing occurs without an associated BY statement</td>
<td></td>
</tr>
<tr>
<td>&quot;OBS= System Option&quot; on page 1572</td>
<td>Specifies when to stop processing observations or records</td>
<td></td>
</tr>
<tr>
<td>&quot;REPLACE System Option&quot; on page 1590</td>
<td>Controls whether you can replace permanently stored SAS data sets</td>
<td></td>
</tr>
<tr>
<td>&quot;REUSE= System Option&quot; on page 1591</td>
<td>Specifies whether or not SAS reuses space when observations are added to a compressed SAS data set</td>
<td></td>
</tr>
<tr>
<td>&quot;UTILLOC= System Option&quot; on page 1620</td>
<td>Specifies one or more file system locations in which applications can store utility files</td>
<td></td>
</tr>
<tr>
<td>Category</td>
<td>SAS System Options</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------------------------</td>
<td>--------------------------------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>&quot;VALIDFMTNAME= System Option&quot; on page 1624</td>
<td>Controls the length for the names of formats and informats that can be used when creating new SAS data sets and format catalogs</td>
</tr>
<tr>
<td></td>
<td>&quot;VALIDVARNAME= System Option&quot; on page 1625</td>
<td>Controls the type of SAS variable names that can be created and processed during a SAS session</td>
</tr>
<tr>
<td>Graphics: Driver settings</td>
<td>&quot;DEVICE= System Option&quot; on page 1508</td>
<td>Specifies a terminal device driver for SAS/GRAPH software</td>
</tr>
<tr>
<td></td>
<td>&quot;GISMAPS= System Option&quot; on page 1536</td>
<td>Specifies the location of the SAS data library that contains SAS/GIS-supplied US Census Tract maps</td>
</tr>
<tr>
<td></td>
<td>&quot;GWINDOW System Option&quot; on page 1536</td>
<td>Controls whether SAS displays SAS/GRAPH output in the GRAPH window of the windowing environment</td>
</tr>
<tr>
<td></td>
<td>&quot;MAPS= System Option&quot; on page 1550</td>
<td>Specifies the locations to search for maps</td>
</tr>
<tr>
<td>Input control: Data Processing</td>
<td>&quot;BYSORTED System Option&quot; on page 1487</td>
<td>Specifies whether observations in one or more data sets are sorted in alphabetic or numeric order or are grouped in another logical order</td>
</tr>
<tr>
<td></td>
<td>&quot;CAPS System Option&quot; on page 1488</td>
<td>Indicates whether to translate input to uppercase</td>
</tr>
<tr>
<td></td>
<td>&quot;CARDIMAGE System Option&quot; on page 1489</td>
<td>Processes SAS source and data lines as 80-byte cards</td>
</tr>
<tr>
<td></td>
<td>&quot;INVALIDDATA= System Option&quot; on page 1542</td>
<td>Specifies the value SAS is to assign to a variable when invalid numeric data are encountered</td>
</tr>
<tr>
<td></td>
<td>&quot;DATESTYLE= System Option&quot; on page 1506</td>
<td>Identifies the sequence of month, day, and year when ANYDTDTE, ANYDTDTM, or ANYDTTME informat data is ambiguous</td>
</tr>
<tr>
<td></td>
<td>&quot;S= System Option&quot; on page 1594</td>
<td>Specifies the length of statements on each line of a source statement and the length of data on lines that follow a DATALINES statement</td>
</tr>
<tr>
<td></td>
<td>&quot;S2= System Option&quot; on page 1596</td>
<td>Specifies the length of secondary source statements</td>
</tr>
<tr>
<td></td>
<td>&quot;SEQ= System Option&quot; on page 1598</td>
<td>Specifies the length of the numeric portion of the sequence field in input source lines or datalines</td>
</tr>
<tr>
<td></td>
<td>&quot;SPOOL System Option&quot; on page 1606</td>
<td>Controls whether SAS writes SAS statements to a utility data set in the WORK data library</td>
</tr>
<tr>
<td></td>
<td>&quot;YEARCUTOFF= System Option&quot; on page 1630</td>
<td>Specifies the first year of a 100-year span that will be used by date informats and functions to read a two-digit year</td>
</tr>
<tr>
<td>Log and procedure output control: ODS printing</td>
<td>&quot;BINDING= System Option&quot; on page 1480</td>
<td>Specifies the binding edge to a printer</td>
</tr>
<tr>
<td></td>
<td>&quot;BOTTOMMARGIN= System Option&quot; on page 1481</td>
<td>Specifies to a printer the size of the margin at the bottom of the page</td>
</tr>
<tr>
<td></td>
<td>&quot;COLLATE System Option&quot; on page 1499</td>
<td>Specifies the collation of multiple copies for output to a printer</td>
</tr>
<tr>
<td>Category</td>
<td>SAS System Options</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Specifying color printing to a printer, if color printing is supported</td>
<td>“COLORPRINTING System Option” on page 1500</td>
<td></td>
</tr>
<tr>
<td>Specifies the number of copies to make when printing to a printer</td>
<td>“COPIES= System Option” on page 1502</td>
<td></td>
</tr>
<tr>
<td>Specifies duplexing controls to a printer</td>
<td>“DUPLEX System Option” on page 1519</td>
<td></td>
</tr>
<tr>
<td>Specifies to a printer the size of the margin on the left side of the page</td>
<td>“LEFTMARGIN= System Option” on page 1544</td>
<td></td>
</tr>
<tr>
<td>Specifies the paper orientation to use when printing to a printer</td>
<td>“ORIENTATION= System Option” on page 1578</td>
<td></td>
</tr>
<tr>
<td>Specifies to a printer the printer bin to receive printed output</td>
<td>“PAPERDEST= System Option” on page 1581</td>
<td></td>
</tr>
<tr>
<td>Specifies to a printer the paper size to use</td>
<td>“PAPERSIZE= System Option” on page 1582</td>
<td></td>
</tr>
<tr>
<td>Specifies to a printer the paper bin to use for printing</td>
<td>“PAPERSOURCE= System Option” on page 1583</td>
<td></td>
</tr>
<tr>
<td>Specifies to a printer the type of paper to use for printing</td>
<td>“PAPERTYPE= System Option” on page 1584</td>
<td></td>
</tr>
<tr>
<td>The PRINTERPATH= option controls which Universal Printing printer will be used for printing.</td>
<td>“PRINTERPATH= System Option” on page 1586</td>
<td></td>
</tr>
<tr>
<td>Specifies the size of the margin at the right side of the page for printed output directed to the ODS printer destination</td>
<td>“RIGHTMARGIN= System Option” on page 1592</td>
<td></td>
</tr>
<tr>
<td>Specifies the location of textures and images that are used by ODS styles</td>
<td>“TEXTURELOC= System Option” on page 1614</td>
<td></td>
</tr>
<tr>
<td>Specifies the size of the margin at the top of the page for the ODS printer destination</td>
<td>“TOPMARGIN= System Option” on page 1617</td>
<td></td>
</tr>
<tr>
<td>Specifies whether to enable Universal Printing services</td>
<td>“UNIVERSALPRINT System Option” on page 1619</td>
<td></td>
</tr>
<tr>
<td>Controls whether BY lines are printed above each BY group</td>
<td>“BYLINE System Option” on page 1486</td>
<td></td>
</tr>
<tr>
<td>Controls alignment of SAS procedure output</td>
<td>“CENTER System Option” on page 1492</td>
<td></td>
</tr>
<tr>
<td>Specifies the default output formatting characters</td>
<td>“FORMCHAR= System Option” on page 1533</td>
<td></td>
</tr>
<tr>
<td>Specifies a character to delimit page breaks in SAS output</td>
<td>“FORMDLIM= System Option” on page 1534</td>
<td></td>
</tr>
<tr>
<td>Determines whether SAS procedures can use labels with variables</td>
<td>“LABEL System Option” on page 1543</td>
<td></td>
</tr>
<tr>
<td>Resets the page number</td>
<td>“PAGENO= System Option” on page 1580</td>
<td></td>
</tr>
<tr>
<td>Category</td>
<td>SAS System Options</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>“PRINTINIT System Option” on page 1587</strong></td>
<td>Initializes the SAS file</td>
<td></td>
</tr>
<tr>
<td><strong>“SKIP= System Option” on page 1600</strong></td>
<td>Specifies the number of lines to skip at the top of each page of SAS output</td>
<td></td>
</tr>
<tr>
<td><strong>“SYSPRINTFONT= System Option” on page 1610</strong></td>
<td>Sets the font for the current default printer</td>
<td></td>
</tr>
<tr>
<td><strong>“DATE System Option” on page 1506</strong></td>
<td>Prints the date and time that the SAS session was initialized</td>
<td></td>
</tr>
<tr>
<td><strong>“DETAILS System Option” on page 1507</strong></td>
<td>Specifies whether to include additional information when files are listed in a SAS data library</td>
<td></td>
</tr>
<tr>
<td><strong>“DTRESET System Option” on page 1518</strong></td>
<td>Updates date and time in the SAS log and in the listing file</td>
<td></td>
</tr>
<tr>
<td><strong>“LINESIZE= System Option” on page 1545</strong></td>
<td>Specifies the line size of SAS procedure output</td>
<td></td>
</tr>
<tr>
<td><strong>“MISSING= System Option” on page 1565</strong></td>
<td>Specifies the character to print for missing numeric values</td>
<td></td>
</tr>
<tr>
<td><strong>“NUMBER System Option” on page 1571</strong></td>
<td>Controls the printing of page numbers</td>
<td></td>
</tr>
<tr>
<td><strong>“PAGEBREAKINITIAL System Option” on page 1579</strong></td>
<td>Begins the SAS log and listing files on a new page</td>
<td></td>
</tr>
<tr>
<td><strong>“PAGESIZE= System Option” on page 1581</strong></td>
<td>Specifies the number of lines that compose a page of SAS output</td>
<td></td>
</tr>
<tr>
<td><strong>“CPUID System Option” on page 1504</strong></td>
<td>Specifies whether hardware information is written to the SAS log</td>
<td></td>
</tr>
<tr>
<td><strong>“ECHOAUTO System Option” on page 1519</strong></td>
<td>Controls whether autoexec code in an input file is echoed to the log</td>
<td></td>
</tr>
<tr>
<td><strong>“LOGPARM= System Option” on page 1546</strong></td>
<td>Controls when SAS log files are opened, closed, and, in conjunction with the LOG= system option, how they are named</td>
<td></td>
</tr>
<tr>
<td><strong>“MSGLEVEL= System Option” on page 1567</strong></td>
<td>Controls the level of detail in messages that are written to the SAS log</td>
<td></td>
</tr>
<tr>
<td><strong>“NOTES System Option” on page 1570</strong></td>
<td>Writes notes to the SAS log</td>
<td></td>
</tr>
<tr>
<td><strong>“OVP System Option” on page 1579</strong></td>
<td>Overprints output lines</td>
<td></td>
</tr>
<tr>
<td><strong>“PRINTMSGLIST System Option” on page 1588</strong></td>
<td>Controls the printing of extended lists of messages to the SAS log</td>
<td></td>
</tr>
<tr>
<td><strong>“SOURCE System Option” on page 1604</strong></td>
<td>Controls whether SAS writes source statements to the SAS log</td>
<td></td>
</tr>
<tr>
<td><strong>“SOURCE2 System Option” on page 1605</strong></td>
<td>Controls whether SAS writes secondary source statements from included files to the SAS log</td>
<td></td>
</tr>
<tr>
<td>Category</td>
<td>SAS System Options</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Macro: SAS macro</td>
<td>“CMDMAC System Option” on page 1495</td>
<td>Determines whether the macro processor recognizes a command-style macro invocation</td>
</tr>
<tr>
<td></td>
<td>“IMPLMAC System Option” on page 1539</td>
<td>Controls whether SAS allows statement-style macro calls</td>
</tr>
<tr>
<td></td>
<td>“MACRO System Option” on page 1550</td>
<td>Specifies whether the SAS macro language is available</td>
</tr>
<tr>
<td></td>
<td>“MAUTOLOCDISPLAY System Option” on page 1550</td>
<td>Displays the source location of the autocall macros in the log when the autocall macro is invoked</td>
</tr>
<tr>
<td></td>
<td>“MAUTOSOURCE System Option” on page 1551</td>
<td>Determines whether the macro autocall feature is available</td>
</tr>
<tr>
<td></td>
<td>“MCOMPILENOTE= System Option” on page 1551</td>
<td>Issues a NOTE to the log upon successful compilation of a macro</td>
</tr>
<tr>
<td></td>
<td>“MERROR System Option” on page 1552</td>
<td>Controls whether SAS issues a warning message when a macro-like name does not match a macro keyword</td>
</tr>
<tr>
<td></td>
<td>“MFILE System Option” on page 1565</td>
<td>Specifies whether MPRINT output is directed to an external file</td>
</tr>
<tr>
<td></td>
<td>“MINDELIMITER= System Option” on page 1565</td>
<td>Specifies the character to be used as the delimiter for the macro IN operator</td>
</tr>
<tr>
<td></td>
<td>“MLOGIC System Option” on page 1566</td>
<td>Controls whether SAS traces execution of the macro language processor</td>
</tr>
<tr>
<td></td>
<td>“MLOGICNEST System Option” on page 1566</td>
<td>Displays macro nesting information in the MLOGIC output in the SAS log</td>
</tr>
<tr>
<td></td>
<td>“MPRINT System Option” on page 1567</td>
<td>Displays SAS statements that are generated by macro execution</td>
</tr>
<tr>
<td></td>
<td>“MPRINTNEST System Option” on page 1567</td>
<td>Displays macro nesting information in the MPRINT output in the SAS log</td>
</tr>
<tr>
<td></td>
<td>“MRECALL System Option” on page 1567</td>
<td>Controls whether SAS searches the autocall libraries for a file that was not found during an earlier search</td>
</tr>
<tr>
<td></td>
<td>“MSTORED System Option” on page 1568</td>
<td>Determines whether the macro facility searches a specific catalog for a stored, compiled macro</td>
</tr>
<tr>
<td></td>
<td>“MSYMTABMAX= System Option” on page 1569</td>
<td>Specifies the maximum amount of memory that is available to macro variable symbol tables</td>
</tr>
<tr>
<td></td>
<td>“MVARSIZE= System Option” on page 1570</td>
<td>Specifies the maximum size for macro variables that are stored in memory</td>
</tr>
<tr>
<td></td>
<td>“SASAUTOS= System Option” on page 1596</td>
<td>Specifies the autocall macro library</td>
</tr>
<tr>
<td></td>
<td>“SASMSTORE= System Option” on page 1597</td>
<td>Specifies the libref of a SAS data library that contains a catalog of stored, compiled SAS macros</td>
</tr>
<tr>
<td></td>
<td>“ERROR System Option” on page 1599</td>
<td>Controls whether SAS issues a warning message when a defined macro variable reference does not match a macro variable</td>
</tr>
<tr>
<td>Category</td>
<td>SAS System Options</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>“SYMBOLGEN System Option” on page 1608</td>
<td>Controls whether the results of resolving macro variable references are written to the SAS log</td>
<td></td>
</tr>
<tr>
<td>“SORTDUP= System Option” on page 1601</td>
<td>Controls the SORT procedure’s application of the NODUP option to physical or logical records</td>
<td></td>
</tr>
<tr>
<td>“SORTEQUALS System Option” on page 1602</td>
<td>Controls how PROC SORT orders observations with identical BY values in the output data set</td>
<td></td>
</tr>
<tr>
<td>“SORTSEQ= System Option” on page 1603</td>
<td>Specifies a language-specific collation sequence for the SORT procedure to use in the current SAS session</td>
<td></td>
</tr>
<tr>
<td>“SORTSIZE= System Option” on page 1603</td>
<td>Specifies the amount of memory that is available to the SORT procedure</td>
<td></td>
</tr>
<tr>
<td>“SETINIT System Option” on page 1599</td>
<td>Controls whether site license information can be altered</td>
<td></td>
</tr>
<tr>
<td>“SORTSIZE= System Option” on page 1603</td>
<td>Specifies the amount of memory that is available to the SORT procedure</td>
<td></td>
</tr>
<tr>
<td>“SUMSIZE= System Option” on page 1607</td>
<td>Specifies a limit on the amount of memory that is available for data summarization procedures when class variables are active</td>
<td></td>
</tr>
<tr>
<td>“ARMAGENT= System Option” on page 1459</td>
<td>Specifies another vendor’s ARM agent, which is an executable module that contains a vendor’s implementation of the ARM API</td>
<td></td>
</tr>
<tr>
<td>“ARMLOC= System Option” on page 1460</td>
<td>Specifies the location of the ARM log</td>
<td></td>
</tr>
<tr>
<td>“ARMSUBSYS= System Option” on page 1461</td>
<td>Enables and disables the ARM subsystems that determine the internal SAS processing transactions to be logged</td>
<td></td>
</tr>
<tr>
<td>“CMPOPT= System Option” on page 1497</td>
<td>Specifies the type of code generation optimizations to use in the SAS language compiler</td>
<td></td>
</tr>
<tr>
<td>“CPUCOUNT= System Option” on page 1503</td>
<td>Specifies the number of processors that the thread-enabled applications should assume will be available for concurrent processing</td>
<td></td>
</tr>
<tr>
<td>“THREADS System Option” on page 1615</td>
<td>Specifies that SAS use threaded processing if it is available</td>
<td></td>
</tr>
</tbody>
</table>

**Dictionary**

**APPLETLOC= System Option**

Specifies the location of Java applets.

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Environment control: Files
PROC OPTIONS GROUP= ENVFILES

Syntax
APPLETLOC="base-URL"

Syntax Description

“base-URL”
specifies the address where the SAS Java applets are located. The maximum address length is 256 characters.

Details
The APPLETLOC= system option specifies the base location (typically a URL) of Java applets. These applets are typically accessed from an intranet server or a local CD-ROM.

Examples
Some examples of the base-URL are
- “file://e:\java"
- "http://server.abc.com/SAS/applets"

ARMAGENT= System Option

Specifies another vendor’s ARM agent, which is an executable module that contains a vendor’s implementation of the ARM API

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window
Category: System administration: Performance
Restriction: After you enable the ARM subsystem, you cannot specify a different ARM agent using ARMAGENT=.

PROC OPTIONS GROUP= PERFORMANCE
See: ARMAGENT= System Option in the documentation for your operating environment.

Syntax
ARMAGENT=module
Syntax Description

module

is the name of the module that contains an ARM agent, which is a program module that contains a vendor's implementation of the ARM API.

Operating Environment Information: The maximum length for the module name is specific to your operating environment. For many operating environments, the maximum length is 32 characters. For the z/OS operating environment, see the SAS companion for the z/OS operating environment.

Default: none

Details

An ARM agent is an executable module that contains a vendor's implementation of the ARM API. The ARM agent is a set of executable routines that are called from an application. The ARM agent and SAS run concurrently. The SAS application passes transaction information to the ARM agent, which collects and manages the writing of the ARM records to the ARM log. SAS, as well as other vendors, provide an ARM agent.

By default, SAS uses the SAS ARM agent. Use ARMAGENT= to specify another vendor's ARM agent in order to monitor both the internal SAS processing transactions (using ARMSUBSYS=) as well as for user-defined transactions (using ARM macros).

See Also


System Options:

“ARMLOC= System Option” on page 1460
“ARMSUBSYS= System Option” on page 1461

ARMLOC= System Option

Specifies the location of the ARM log

Valid in: configuration file, SAS invocation, OPTIONS statement, System Options window

Category: System administration: Performance

PROC OPTIONS GROUP= PERFORMANCE

Syntax

ARMLOC=fileref | filename
Syntax Description

\textit{fileref}  
is a SAS name that is associated with the physical location of the ARM log. To assign a fileref, use the \texttt{FILENAME} statement.

\textit{filename}  
is the physical location of the log. Include the complete pathname and the filename. You can use either single or double quotation marks.

\textbf{Default:} Filename: ARMLOG.LOG

\textbf{Restriction:} For all operating environments except z/OS, if you specify the \texttt{ARMLOC=} system option in your configuration file, you must specify the filename, not a fileref.

Details

The ARM log is an external output text file that contains the logged ARM transaction records. The ARM log gathers transaction information for both the internal SAS processing transactions (depending on the value of the \texttt{ARMSUBSYS=} system option) as well as for user-defined transactions (using ARM macros).

You can change the location of the ARM log after initializing an ARM subsystem, but records that were written to the old ARM log are not copied to the new ARM log. Therefore, you should issue \texttt{ARMLOC=} before issuing the first \texttt{ARMSUBSYS=} so that all records are written to the same ARM log.

See Also


System Options:

“\texttt{ARMAGENT=} System Option” on page 1459

“\texttt{ARMSUBSYS=} System Option” on page 1461

\begin{verbatim}
ARMSUBSYS= System Option
\end{verbatim}

\textbf{Enables and disables the ARM subsystems that determine the internal SAS processing transactions to be logged}

\textbf{Valid in:} configuration file, SAS invocation, \texttt{OPTIONS} statement, SAS System Options window

\textbf{Category:} System administration: Performance

\textbf{PROC OPTIONS GROUP=} PERFORMANCE

\textbf{Restriction:} After you enable the ARM subsystems, you cannot specify a different ARM agent using \texttt{ARMAGENT=}.

\textbf{Default:} (ARM\_NONE)
Syntax

ARMSUBSYS=(ARM_NONE | ARM_ALL | subsystem1 <item1 <item2 <...> > > <, subsystem2 <item1 <item2 <...> > > ><OFF>)

Syntax Description

ARM_NONE
specifies that no internal SAS processing transactions are written to the ARM log. This is the default setting.

ARM_ALL
specifies that all available internal SAS processing transactions are written to the ARM log.

subsystem
specifies an ARM subsystem, which is a group of internal SAS processing transactions that are to be written to the ARM log. The available subsystems are

ARM_DSIO
collects SAS data set I/O processing information.

ARM_IOM
collects IOM Server processing information.

ARM_PROC
collects SAS procedure and DATA step processing information.

ARM_OAP_SESSION
collects SAS OLAP Server session information.

item
specifies a name that determines the type and amount of transaction logging for each subsystem. Use the item specification(s) as a filtering technique so that only the details that you are interested in are logged. For example, if you want only one type of transaction, list the single item, or if you want multiple transactions for a subsystem, list each item that you want. Items are associated with each subsystem as follows:

ARM_DSIO

OPENCLOSE
logs a SAS data set open and close transaction as a start record when a data set is opened and a stop record when it is closed.

VARDEF
logs OPENCLOSE records and an update record for each defined variable (output opens).

VARSEL
logs OPENCLOSE records and an update record for each selected variable (input and update opens).

VARINFO
logs OPENCLOSE, VARDEF, and VARSEL records.

WHEREOPT
logs OPENCLOSE records and an update record for each index that is selected as a result of WHERE processing optimization. This update is available for the default Base SAS engine and the V6 compatibility engine only.
WHEREINFO
   logs OPENCLOSE, WHEREOPT, and WHERETXT records.

WHEREINFO
   logs OPENCLOSE records and one or more update records that contain a
textual representation of the active WHERE expression. Each record can hold
approximately 1,000 bytes.

MIN
   logs the minimum amount of information. For SAS Version 9, MIN logs the
OPENCLOSE records.

MAX
   logs the maximum amount of information. For SAS Version 9, MAX logs all of
the ARM_DSIO records. This is the default for ARM_DSIO.

LEVEL1
   logs OPENCLOSE, VARDEF, and VARSEL records.

LEVEL2
   logs LEVEL1, WHEREOPT, and WHERETXT records.
   For more information about the logged records, see “Understanding the Records
Written by the ARM_DSIO Subsystem” on page 1464.

ARM_IOM
   logs records for IOM server methods to monitor performance of IOM servers. For
more information about the logged records, see “Understanding the Records
Written by the ARM_IOM Subsystem” on page 1466.

ARM_PROC
   logs PROC and DATA step execution time (that is, when the step begins and when
it terminates) as a start and a stop record. For more information about the logged
records, see “Understanding the Records Written by the ARM_PROC Subsystem”
on page 1470.

ARM_OOLAP_SESSION
   OLAP_SESSION
      logs initialization and termination records in order to tell you which server was
used and for how long. This is the default transaction for the
   ARM_OOLAP_SESSION subsystem.

MDX_QUERY
   logs a record for each query sent. This transaction stores the cube name and the
size of the result set in cells, which are additional identifiers or metrics.

DATA_QUERY
   logs a record for each region execution, that is, each individual data retrieval
from stored cube aggregations or from the cache. The transaction class stores
the following additional identifiers or metrics:
      region ID
      aggregate ID
      number of returned records
      source type (MOLAP, CSAS, CACHE, CSPDS)

   The DATA_QUERY transaction serves as the primary input for both automatic
   and manual cube optimization.

MDX_STRING
   logs a record for the MDX_QUERY transaction that contains the actual MDX
query string.
For more Information about the logged records, see “Understanding the Records Written by the ARM_OLAP_SESSION Subsystem” on page 1470.

**OFF**
disables the specified subsystem. For example, in the following code, all subsystems are enabled for the DATA step, and then the ARM_PROC subsystem is disabled for the PRINT procedure:

```plaintext
options armsubsys=(arm_all);
data a;
    x=1;
run;

options armsubsys=(arm_proc off);
proc print;
run;
```

**Details**

**Overview of ARM Subsystems**  The ARMSUBSYS= system option enables and disables the ARM subsystems, which determine the internal SAS transaction processing for which you want to monitor. An ARM subsystem is a group of internal SAS processing transactions such as DATA and PROC step processing and file input/output processing.

If you want to specify a different ARM log location by using the ARMLOC= system option, be sure to issue ARMLOC= before you enable an ARM subsystem so that the subsystem initialization record is written to the new log.

*Note:* There is a subsystem for collecting memory usage information. For more information about that subsystem, contact SAS Technical Support.

**Understanding the Records Written by the ARM_DSIO Subsystem**  The ARM_DSIO subsystem writes records to the ARM log that collects SAS data set input/output processing information. These are the records that are written to the ARM log:

- **I (initialization) record** is an initialization record, with one record written per session when the ARM subsystem is initialized. It starts with an I, followed by these items:
  - the SAS datetime value for session start
  - an application ID
  - a user start time
  - a system start time
  - an application name
  - a user ID.

**Output:**

```
I,1326479452.427000,1,1.171684,1.532203,SAS,sasabc
```

- **G (GetID) record** is a transaction ID record, with one record written per transaction. It starts with a G, followed by these items:
  - the SAS datetime value when the record was written
  - an application ID from the I record
a transaction class ID
- a transaction name
- a transaction description
- a description of the values provided in subsequent S (start) and P (stop) records.

**Output:**

```
G,1326479452.447000,1,1,MVA_DSIO.OPEN_CLOSE,DATA SET OPEN/CLOSE,
LIBNAME,ShortStr,MEMTYPE,ShortStr,MEMNAME,LongStr
```

LIBNAME refers to the libref for a SAS data library, MEMTYPE refers to the member type (either DATA or VIEW), and MEMNAME refers to a SAS data set name.

**S (start) record**

is a start record, with one record written each time a file is opened. It starts with an S, followed by these items:

- the SAS datetime value when the record was written
- an application ID from the I record
- a transaction class ID from the G record
- a transaction ID
- a user start time
- a system start time
- the actual libref, member type, and member name of the opened file.

**Output:**

```
S,1326479486.396000,1,1,1,1.311886,2.22908,WORK ,DATA ,GRADES
```

**P (stop) record**

is a stop record, with one record written each time a file is closed. It starts with a P, followed by these items:

- the SAS datetime value when the record was written
- an application ID from the I record
- a transaction class ID from the G record
- a transaction ID from the associated S record
- a user start time
- a system start time
- the actual libref, member type, and member name of the closed file.

**Output:**

```
P,1326479486.706000,1,1,1,1.331915,2.22908,0,WORK ,DATA ,GRADES
```

**U (update) record**

is an update record, with one record written each time a variable is defined or selected, and each time an index is used during WHERE processing optimization. A U record displays the text of a WHERE expression. It starts with a U, followed by these items:

- the SAS datetime value when the record was written
For variable definition and selection, the variable type is specified with a 1 for a numeric variable or a 2 for a character variable, then the name of the variable, followed by DEF for definition or SEL for selection:

**Output:**

```
U,1326479486.406000,1,1,1,1.321900,2.22908,2,VAR(2,Student),DEF
U,1326479508.508000,1,1,2,1.612318,2.443513,2,VAR(2,Student),SEL
```

For index selection, the index type is specified with an S for simple or a C for complex, followed by the name of the index:

```
U,1326479606.48000,1,1,4,2.403456,3.915630,2,INDEX(S,Test1),SEL
```

For WHERE expression text information, the expression is specified

```
U,1326479606.48000,1,1,4,2.403456,3.915630,2,WHERE(0),test1>60
```

The ARM_IOM subsystem writes records to the ARM log that collects IOM server method processing information. These records are written to the ARM log:

**I (initialization) record**

is an initialization record, with one record written per session when the ARM subsystem is initialized. It starts with an I, followed by these items:

- the SAS datetime value for session start
- an application ID
- a user start time
- a system start time
- an application name
- a user ID.

**Output:**

```
I,1327874323.742000,1,0.220316,0.320460,SAS,sasxxx
```
G (GetID) record
is a transaction ID record, with one record written for each method and attribute and one record written for creating the correlator. It starts with a G, followed by

- the SAS datetime value when the record was written
- an application ID from the I record
- a transaction class ID
- method and attribute transaction names.

Method and attribute transactions are named component name::method/attribute name, or, if the interface and component names are not the same, component name::interface name::method/attribute name:

G,1327874341.77000,1,2,TestTypes::SetParent,

S (start) record
is a start record, with one record written at the beginning of each method or attribute. If a correlator is available, it is placed on the transaction. (The correlator is created by a separate start transaction on the transaction defined for that purpose.) The record starts with an S, followed by these items:

- the SAS datetime value when the record was written
- an application ID
- a transaction class ID
- a transaction ID
- a user start time, and
- a system start time.

Output:

S,1327874344.1000,1,1,1,0.620892,0.741065

P (stop) record
is a stop record, with one record written at the end of each method or attribute. It starts with a P, followed by these items:

- the SAS datetime value when the record was written
- an application ID
- a transaction class ID
- a transaction ID from the associated S record
- a user start time
- a system start time.

Output:

P,1327874348.658000,1,9,2,0.620892,0.741065,0
Here is an ARM log for the ARM_IOM subsystem:
Output 8.4 ARM Log for ARM_IOM Subsystem

I,1327874323.742000,1,0.220316,0.320460,SAS,sasmaa
G,1327874328.280000,1,1,IOMCOMPTrans,Transaction for creating correlator
G,1327874341.770000,1,2,TestTypes::SetParent,
G,1327874341.770000,1,3,TestTypes::TypeBoolean,
G,1327874341.770000,1,4,TestTypes::DateTime,
G,1327874341.770000,1,5,TestTypes::Double,
G,1327874341.770000,1,6,TestTypes::Enum,
G,1327874341.770000,1,7,TestTypes::TypeFloat,
G,1327874341.770000,1,8,TestTypes::Interface,
G,1327874341.770000,1,9,TestTypes::TypeLong,
G,1327874341.770000,1,10,TestTypes::Octet,
G,1327874341.770000,1,11,TestTypes::TypeShort,
G,1327874341.770000,1,12,TestTypes::String,
G,1327874341.770000,1,13,TestTypes::UUID,
G,1327874341.770000,1,14,TestTypes::TypeEnum,
G,1327874341.770000,1,15,TestTypes::TypeAny,
G,1327874341.770000,1,16,TestTypes::TypeBoolean1dArray,
G,1327874341.770000,1,17,TestTypes::TypeDateTime1dArray,
G,1327874341.770000,1,18,TestTypes::TypeDouble1dArray,
G,1327874341.770000,1,19,TestTypes::TypeFloat1dArray,
G,1327874341.770000,1,20,TestTypes::TypeLong1dArray,
G,1327874341.770000,1,21,TestTypes::TypeOctet1dArray,
G,1327874341.770000,1,22,TestTypes::TypeShort1dArray,
G,1327874341.770000,1,23,TestTypes::TypeString1dArray,
G,1327874341.770000,1,24,TestTypes::TypeUUID1dArray,
G,1327874341.870000,1,25,TestTypes::TypeAny1dArray,
G,1327874341.870000,1,26,TestTypes::TypeBoolean2dArray,
G,1327874341.870000,1,27,TestTypes::TypeDateTime2dArray,
G,1327874341.870000,1,28,TestTypes::TypeDouble2dArray,
G,1327874341.870000,1,29,TestTypes::TypeEnum2dArray,
G,1327874341.870000,1,30,TestTypes::TypeFloat2dArray,
G,1327874341.870000,1,31,TestTypes::TypeLong2dArray,
G,1327874341.870000,1,32,TestTypes::TypeOctet2dArray,
G,1327874341.870000,1,33,TestTypes::TypeShort2dArray,
G,1327874341.870000,1,34,TestTypes::TypeString2dArray,
G,1327874341.870000,1,35,TestTypes::TypeUUID2dArray,
G,1327874341.870000,1,36,TestTypes::TypeAny2dArray,
G,1327874341.870000,1,37,TestTypes::TestMult Name Get,
G,1327874341.870000,1,38,TestTypes::TestMult Test

S,1327874344.1000,1,1,0.620892,0.741065
C,1327874348.658000,1,9,2,1,0.620892,0.741065
P,1327874348.658000,1,9,2,0.620892,0.741065,0
P,1327874348.658000,1,1,1,0.620892,0.741065,0
S,1327874348.688000,1,1,3,0.620892,0.741065
C,1327874348.688000,1,3,4,1,3,0.620892,0.741065
P,1327874348.688000,1,3,4,0.620892,0.741065,0
P,1327874348.688000,1,1,3,0.620892,0.741065,0
Understanding the Records Written by the ARM_PROC Subsystem

The ARM_PROC subsystem writes records to the ARM log that collects SAS procedure and DATA step processing information. These are the records written to the ARM log:

S (start) record

is a start record, with one record written immediately before the procedure starts execution. It starts with an S, followed by these items:

- the SAS datetime value when the record was written
- an application ID
- a transaction class ID
- a transaction ID
- a user start time
- a system start time
- the procedure or DATA step name.

Output:

```
S, 1323716628.571000, 1, 1, 2, 2.834075, 8.482196, PRINT
```

P (stop) record

is a stop record, with one record written when the procedure terminates. It starts with a P, followed by these items:

- the SAS datetime value when the record was written
- an application ID
- a transaction class ID
- a transaction ID from the associated S record
- a user start time
- a system start time.

Output:

```
P, 1323716633.398000, 1, 1, 2, 2.944233, 8.772614, 0
```

Understanding the Records Written by the ARM_OLAP_SESSION Subsystem

The ARM_OLAP_SESSION subsystem writes records to the ARM log that collects information about a SAS OLAP Server session. A SAS OLAP Server is started as a system process, waiting for clients to connect to it. Each client connection establishes a session on the server. Each session can then serve MDX queries. Each MDX query is typically split into one or more regions, which translate calls against the cube’s data, the data queries.

These records are written for the ARM_OLAP_SESSION subsystem. The initialization and termination records give you summary information on each SAS OLAP Server invocation:

I (initialization) record

is an initialization record, with one record written per OLAP Server invocation when the ARM subsystem is initialized. It starts with an I, followed by these items:

- the SAS datetime value of server start
- an application ID
- a user CPU (start) time
- a system CPU (start) time
an application name OLAP_SERVER.

**Output:**

\[ I,1320592800.822000,2,0.380547,0.630907,OLAP_SERVER, \]

E (end) record

is a termination record, with one record written per OLAP Server termination. It starts with an E, followed by these items:

- the SAS datetime value of the server termination
- an application ID from the I record
- a user CPU (stop) time
- a system CPU (stop) time.

**Output:**

\[ E,1320592802.805000,2,1.281843,0.791137 \]

The E records are written for the OLAP_SESSION transaction, which records each session that is started by a client connection. The E records provide the user ID of the client user that started the session:

G (GetID) record

is an OLAP_SESSION transaction record, with one record written per OLAP Server invocation. It starts with a G, followed by these items:

- the SAS datetime value when the record was written
- an application ID from the I record
- a transaction class ID
- a transaction class name OLAP_SESSION
- a transaction class description OLAP Session
- a description of the values that are provided in subsequent S (start) and P (stop) records.

**Output:**

\[ G,1337615817.801000,2,1,OLAP_SESSION,OLAP Session,User Name,LongStr \]

User Name is the ID of the client user that started the session.

S (start) record

is a start record, with one record written for each session. It starts with an S, followed by these items:

- the SAS datetime value when the session started
- an application ID from the I record
- a transaction class ID from the G record
- a transaction ID
- a user CPU (start) time
- a system CPU (start) time
- the user ID of the client user, if available.

**Output:**

\[ S,1337615818.502000,2,1,2,52952,0.901296,sasabc \]
P (stop) record
is a stop record, with one record written for each session. It starts with a P,
followed by these items:

- the SAS datetime value when the session ended
- an application ID from the I record
- a transaction class ID from the G record
- a transaction ID from the associated S record
- a user CPU (stop) time
- a system CPU (stop) time
- the status 0=OK.

Output:

P,1337615819.383000,2,1,2,1.13038,0.931339,0

U (update) record
is an update record, with one record written for each hierarchy for each cube. The "U" record for the OLAP_SESSION transaction is only written when the DATA_QUERY transaction is used. It starts with a U, followed by these items:

- the SAS datetime value when the record was written
- an application ID from the I record
- a transaction class ID from the G record
- a transaction ID from the S record
- a user CPU time
- a system CPU time
- a buffer type (always 2, which indicates that 1,024 bytes of text will follow)
- a string with the following format:
  - char(32) cube name
  - char(4) hierarchy number—used to refer to the hierarchy later in the DATA_QUERY update records that identify regions and aggregations
  - char(32) hierarchy name
  - char(4) number of hierarchy levels.

Output:

U,1355324046.943000,2,1,2,1.625000,2.15625,2,SALES 1CUSTOMER 4
U,1355324046.943000,2,1,2,1.625000,2.15625,2,SALES 2PRODUCT 5
U,1355324046.943000,2,1,2,1.625000,2.15625,2,SALES 3TIME 4

These records are written for the MDX_QUERY transaction, which records each query that is sent. These records provide the cube name and the size of the result set in cells:

G (GetID) record
is an MDX_QUERY transaction record, with one record written per OLAP server invocation. It starts with a G, followed by these items:

- the SAS datetime value when the record was written
- an application ID from the I record
- a transaction class ID
- a transaction name MDX_QUERY
- a transaction description MDX Query
SAS System Options △ ARMSUBSYS= System Option 1473

- a description of the values that are provided in subsequent S or C (start) and P (stop) records.

Output:

```
G,1320594857.747000,3,2,MDX_QUERY,MDX Query,Result Set Size,Gauge32,
Cube Name,LongStr
```

Result Set Size refers to the size of the returned query in cells and Cube Name is the name of the cube being queried.

S or C (start) record
is a start record, with one record written for each query. It starts with an S, followed by these items:

- the SAS datetime value when the query started
- the application ID from the I record
- the transaction class ID from the G record
- a transaction ID
- the user CPU (start) time
- the system CPU (start) time.

Output:

```
S,1320594857.787000,3,2,2,1.341929,0.731051
```

If the OLAP_SESSION level was also requested, then the MDX_QUERY record is correlated to its parent session record, and starts with a C, followed by these items:

- the SAS datetime value when the query started
- an application ID from the I record
- a transaction class ID from the G record
- a transaction ID
- the parent transaction class ID
- the parent transaction ID
- a user CPU (start) time
- a system CPU (start) time.

Output:

```
C,1320594857.787000,3,2,2,1,2,1.341929,0.731051
```

P (stop) record
is a stop record, with one record written for each query. It starts with a P, followed by these items:

- the SAS datetime value when the query ended
- an application ID from the I record
- a transaction class ID from the G record
- a transaction ID from the associated S or C record
- a user CPU (stop) time
- a system CPU (stop) time
- a status (0=OK, 2=query failed)
- the size of the returned query in cells
the name of the cube that was queried.

**Output:**

```
P,1320594858.948000,3,2,2,2.52952,0.781123,0,5,MDDBCARS
```

These records are written for the DATA_QUERY transaction, which records each region execution, that is, each individual data retrieval from stored cube aggregations or from the cache. These records provide region IDs, aggregate IDs, the number of returned records, the source type, and the thread index. The DATA_QUERY transaction is the primary input for both automatic and manual cube optimization.

**G (GetID) record**

is a DATA_QUERY transaction identifier record, with one record written per session. It starts with a G, followed by these items:

- the SAS datetime value when the record was written
- the application ID from the I record
- a transaction class ID
- the transaction name DATA_QUERY
- the transaction description Plugin Call
- a description of the values that are provided in subsequent S or C (start) and P (stop) records. They are
  - Query Aggregate
  - Source Aggregate
  - Result Set Size
  - Source Type
  - Thread Index
  - Cube Name.

**Output:**

```
G,1359310645.798000,2,4,DATA_QUERY,Plugin Call,Query Aggregate,Id32,Source Aggregate,Id32,Result Set Size,Gauge32,Source Type,Id32,Thread Index,Gauge32,Cube Name,LongStr
```

**S or C (start subquery) record**

is a start subquery record, with one record written for each data access. It starts with an S, followed by these items:

- the SAS datetime value when the subquery started
- the application ID from the I record
- a transaction class ID from the G record
- a transaction ID
- the CPU (start) time
- the system CPU (start) time.

**Output:**

```
S,1320596204.653000,2,2,2,1.51512,0.630907
```

If the MDX_QUERY level was also requested, then the DATA_QUERY record is correlated to its parent MDX query record, and starts with a C, followed by these items:
- the SAS datetime value when the subquery started
- the application ID from the I record
- the transaction class ID from the G record
- a transaction ID
- the parent transaction class ID
- the parent transaction ID
- the user CPU (start) time
- the system CPU (start) time.

Output:

C,1320596204.653000,2,2,2,1,1,1.51512,0.630907

P (stop subquery) record
is a stop subquery record, with one record written for each data access. It starts with a P, followed by these items:

- the SAS datetime value when the subquery ended
- the application ID from the I record
- the transaction class ID from the G record
- the transaction ID from the S or C record
- the user CPU (stop) time
- the system CPU (stop) time
- a status (0=OK, 2=subquery failed)
- a region sequential number (per Update record)
- an aggregation sequential number (per Update record)
- the size of returned query in records
- the plug-in type (0=CSPDS, 1=CSAS, 2=CACHE, 3=MOLAP)
- the thread index
- the cube name.

Output:

P,1320596205.485000,2,2,2,1.181699,0.670964,0,1,31,5,0,0,SALES

U (update) record
is an update record that is written for each new region and stored aggregation. It starts with a U, followed by these items:

- the SAS datetime value when the record was written
- an application ID from the I record
- a transaction class ID from the G record
- a transaction ID from the S record
- a user CPU time
- a system CPU time
- a buffer type (always 2, which indicates that 1,024 bytes of text will follow)
- a string with the following format:
  - char(32) cube name
  - char(16) unique sequential number, used in the DATA_QUERY Stop record to identify region or aggregation
  - char(4) number of hierarchies in the region or aggregation
repeated for each hierarchy in the region or aggregate:
- char(4) hierarchy number (per OLAP_SESSION Update record)
- char(4) hierarchy level

**Output:**

```
U,1355324092.960000,2,3,61,6.93750,5.734375,2,SALES 4 1 1 1
```

This record is written for the MDX_STRING transaction, which writes an additional record for the MDX_QUERY transaction. The record contains the actual MDX query string.

**U (update) record**

is an update record, with one record written for each query. It starts with a U, followed by these items:

- the SAS datetime value when the record was written
- an application ID from the I record
- a transaction class ID from the G record
- a transaction ID from the S record
- a user CPU time
- a system CPU time
- a buffer type (always 2, which indicates that 1,024 bytes of text will follow)
- the actual MDX string.

**Output:**

```
U,1320589796.262000,2,1,1,0.670964,2,SELECT {
[DATE].[DATEH].[ALL DATE].CHILDREN} ON COLUMNS, {
[MEASURES].[MEASURES].[SALES_SUM]}
ON ROWS FROM MDDBCARS
```

**Examples**

The following example shows the ARM subsystem ARM_DSIO, which collects SAS data set I/O processing information. The OPENCLOSE item logs the SAS data set open and close transaction.

```
options armloc=`myarmlog.txt` armsubsys=(ARM_DSIO OPENCLOSE);
```

The following example shows the ARM subsystem ARM_ALL, which specifies that all available internal SAS processing transactions are written to the ARM log.

```
options
    armagent=sasarmmg
    armsubsys=arm_all
    armloc=mylog;
```

**See Also**


System Options:

“ARMAGENT= System Option” on page 1459
“ARMLOC= System Option” on page 1460
ASYNCHIO System Option

Specifies whether asynchronous I/O is enabled

Valid in: configuration file, SAS invocation
Category: Files: SAS Files
PROC OPTIONS GROUP= SASFILES

Syntax
ASYNCHIO | NOASYNCHIO

Syntax Description

ASYNCHIO
allows other logical SAS tasks to execute (if any are ready) while the I/O is being done. This might improve system performance.

NOASYNCHIO
causes data set I/O to wait for completion. This is the default.

AUTHPROVIDERDOMAIN= System Option

Associates a domain suffix with an authentication provider

Valid in: configuration file, SAS invocation
Alias: AUTHPD
Category: Environment control: Initialization and operation
PROC OPTIONS GROUP= EXECMODES

Syntax
AUTHPROVIDERDOMAIN= (provider-1:domain-1<, ...provider-n:domain-n>)

Syntax Description

provider
specifies the authentication provider that is associated with a domain. These are valid values for provider:

ADIR specifies that the authentication provider be a Microsoft Active Directory that accepts a bind containing user names and passwords for authentication.
HOSTUSER specifies that user names and passwords be authenticated by using the authentication processing that is provided by the host operating system.

*Operating Environment Information:* Under the Windows operating environment, assigning the authentication provider using the HOSTUSER domain is the same as assigning the authentication provider using the AUTHSERVER system option. You might want to use the AUTHPROVIDERDOMAIN= system option when you specify multiple authentication providers.

LDAP specifies that the authentication provider use a directory server to specify the bind distinguished name (BINDDN) and a password for authentication.

domain specifies a site-specific domain name. Quotation marks are required if the *domain* value contains blanks.

**Details**

SAS is able to provide authentication of a user through the use of many authentication providers. The AUTHPROVIDERDOMAIN= system option associates a domain suffix with an authentication provider. This association enables the SAS server to choose the authentication provider by the domain name that is presented.

The maximum length for the AUTHPROVIDERDOMAIN option value is 1,024 characters.

To use the Microsoft Active Directory or LDAP authentication providers, these environment variables must be set in the server or spawner startup script:

**Microsoft Active Directory Server:**

- `AD_PORT=Microsoft Active Directory port number`
- `AD_HOST=Microsoft Active Directory host name`

**LDAP Server:**

- `LDAP_PORT=LDAP port number`
- `LDAP_BASE=base distinguished name`
- `LDAP_HOST=LDAP host name`

**LDAP Server for users connecting with a user ID instead of a distinguished name (DN):**

- `LDAP_PRIV_DN=privileged DN that is allowed to search for users`
- `LDAP_PRIV_PW=LDAP_PRIV_DN password`

*Note:* If the LDAP server allows anonymous binds, then `LDAP_PRIV_DN` and `LDAP_PRIV_PW` are not required.

In addition to setting these environment variables, you can set the LDAP_IDATTR environment variable to the name of the person-entry LDAP attribute that stores the user ID if the attribute does not contain the default value of `uid`.

**Examples**

The following examples show you how to specify the AUTHPROVIDERDOMAIN option:

- `authproviderdomain=(ldap:domain1)`
- `authproviderdomain=(ldap:'my domain')`
□ authpd=(hostuser:'my domain', ldap:domain1)

AUTOSAVELOC= System Option

Specifies the location of the Program Editor autosave file

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Environment control: Display

PROC OPTIONS GROUP= ENVDISPLAY

Syntax

AUTOSAVELOC= “destination”

Syntax Description

destination

specifies the pathname of the autosave file. If destination contains spaces or is specified in an OPTIONS statement, then enclose destination in quotation marks.

BATCH System Option

Specifies whether batch settings for LINESIZE, OVP, PAGESIZE, and SOURCE are in effect when SAS executes

Valid in: configuration file, SAS invocation

Category: Environment control: Initialization and operation

PROC OPTIONS GROUP= EXECMODES

See: BATCH System Option in the documentation for your operating environment.

Syntax

BATCH | NOBATCH

Syntax Description

BATCH

specifies that SAS use the batch settings of LINESIZE=, OVP, PAGESIZE=, and SOURCE. At the start of an interactive SAS session, you can use the BATCH setting to simulate the behavior of the system in batch mode.
NOBATCH

specifies that SAS not use the batch settings for LINESIZE=, OVP, PAGESIZE=, and SOURCE. While in batch mode, you can specify NOBATCH to use the default (nonbatch) settings for the options LINESIZE=, OVP, PAGESIZE=, and SOURCE.

Details

The setting of the BATCH option does not specify the method of operation. BATCH only sets the appropriate batch settings for a collection of options that are in effect when SAS executes.

Operating Environment Information: The default setting for BATCH depends on your operating environment and the SAS method of operation.

BINDING= System Option

Specifies the binding edge to a printer

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Log and procedure output control: ODS printing

PROC OPTIONS GROUP= ODSPRINT

Syntax

BINDING=DEFAULTEDGE | LONDEGE | SHORTEDGE

Syntax Description

DEFAULT | DEFAULTEDGE
specifies that duplexing is done using the default binding edge.

LONG | LONDEGE
specifies the long edge as the binding edge for duplexed output.

SHORT | SHORTEDGE
specifies the short edge as the binding edge for duplexed output.

Details

The binding edge setting determines how the paper is oriented before output is printed on the second side.

Operating Environment Information: Most SAS system options are initialized with default settings when SAS is invoked. However, the default settings and/or option values for some SAS system options may vary both by operating environment and by site. For details, see the SAS documentation for your operating environment.

For additional information on declaring an ODS printer destination, see ODS statements in SAS Output Delivery System: User’s Guide.
For additional information on the SAS universal print facility, see “Printing with SAS” in *SAS Language Reference: Concepts.*

**See Also**

System Option:

“DUPLEX System Option” on page 1519

---

**BOTTOMMARGIN= System Option**

Specifies to a printer the size of the margin at the bottom of the page

**Valid in:** configuration file, SAS invocation, OPTIONS statement, SAS System Options window

**Category:** Log and procedure output control: ODS printing

PROC OPTIONS GROUP= ODSPRINT

**Syntax**

`BOTTOMMARGIN=margin-size | “margin-size [margin-unit]”`

**Syntax Description**

*margin-size*

specifies the size of the margin.

**Restriction:** The bottom margin should be small enough so that the top margin plus the bottom margin is less than the height of the paper.

**Interactions:** Changing the value of this option may result in changes to the value of the PAGESIZE= system option.

*[margin-unit]*

specifies the units for margin-size. The margin-unit can be *in* for inches or *cm* for centimeters.

**Default:** inches

**Requirement:** When you specify margin-unit, enclose the entire option value in double quotation marks.

**Details**

Note that all margins have a minimum that is dependent on the printer and the paper size.

*Operating Environment Information:* Most SAS system options are initialized with default settings when SAS is invoked. However, the default settings and/or option values for some SAS system options may vary both by operating environment and by site. For details, see the SAS documentation for your operating environment.
For additional information on declaring an ODS printer destination see ODS statements in SAS Output Delivery System: User’s Guide.

For additional information on the SAS universal print facility, see “Printing with SAS” in SAS Language Reference: Concepts.

See Also

System Options:
“LEFTMARGIN= System Option” on page 1544
“RIGHTMARGIN= System Option” on page 1592
“TOPMARGIN= System Option” on page 1617

---

**BUFNO= System Option**

Specifies the number of buffers to be allocated for processing SAS data sets

**Valid in:** configuration file, SAS invocation, OPTIONS statement, SAS System Options window

**Category:** Files: SAS Files

**PROC OPTIONS GROUP=** SASFILES

**See:** BUFNO= System Option in the documentation for your operating environment.

---

**Syntax**

BUFNO= MIN | MAX | n | nK | nM | nG | nT | hexX

**Syntax Description**

**MIN**

sets the minimum number of buffers to 0, which causes SAS to use the minimum optimal value for the operating environment. This is the default.

**MAX**

sets the number of buffers to the maximum possible number in your operating environment, up to the largest four-byte, signed integer which is $2^{31} - 1$, or approximately 2 billion.

**CAUTION:**
The recommended maximum for this option is 10. △

**n | nK | nM | nG | nT**

specifies the number of buffers to be allocated in multiples of 1 (bytes); 1,024 (kilobytes); 1,048,576 (megabytes); 1,073,741,824 (gigabytes); or 1,099,511,627,776 (terabytes). For example, a value of 8 specifies 8 bytes, and a value of 3m specifies 3,145,728 bytes.

**hexX**
specifies the number of buffers as a hexadecimal value. You must specify the value beginning with a number (0–9), followed by an X. For example, the value 2dx specifies 45 buffers.

Details

The number of buffers is not a permanent attribute of the data set; it is valid only for the current SAS session or job.

BUFNO= applies to SAS data sets that are opened for input, output, or update. Using BUFNO= can improve execution time by limiting the number of input/output operations that are required for a particular SAS data set. The improvement in execution time, however, comes at the expense of increased memory consumption.

Operating Environment Information: The syntax that is shown here applies to the OPTIONS statement. On the command line or in a configuration file, the syntax is specific to your operating environment. For details, see the SAS documentation for your operating environment.

Comparisons

- You can override the BUFNO= system option by using the BUFNO= data set option.
- To request that SAS allocate the number of buffers based on the number of data set pages and index file pages, use the SASFILE statement.

See Also

Data Set Option:
“BUFNO= Data Set Option” on page 10
System Option:
“BUFSIZE= System Option” on page 1483
Statements:
“SASFILE Statement” on page 1391

BUFSIZE= System Option

Specifies the permanent buffer page size for output SAS data sets

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window
Category: Files: SAS Files
PROC OPTIONS GROUP= SASFILES
See: BUFSIZE= System Option in the documentation for your operating environment.

Syntax

BUFSIZE=n | nK | nM | nG | nT | hexX | MIN | MAX
Syntax Description

\[ n \mid nK \mid nM \mid nG \mid nT \]

specifies the page size in multiples of 1 (bytes); 1,024 (kilobytes); 1,048,576 (megabytes); 1,073,741,824 (gigabytes); or 1,099,511,627,776 (terabytes). For example, a value of \( 8 \) specifies 8 bytes, and a value of \( 3m \) specifies 3,145,728 bytes.

The default is 0, which causes SAS to use the minimum optimal page size for the operating environment.

\[ \text{hexX} \]

specifies the page size as a hexadecimal value. You must specify the value beginning with a number (0–9), followed by an X. For example, the value \( 2dx \) sets the page size to 45 bytes.

\[ \text{MIN} \]

sets the page size to the smallest possible number in your operating environment, down to the smallest four-byte, signed integer, which is \(-2^{31}-1\), or approximately \(-2\) billion bytes.

**CAUTION:** This setting might cause unexpected results and should be avoided. Use BUFSIZE=0 in order to reset the buffer page size to the default value in your operating environment.

\[ \text{MAX} \]

sets the page size to the maximum possible number in your operating environment, up to the largest four-byte, signed integer, which is \(2^{31}-1\), or approximately 2 billion bytes.

Details

The page size is the amount of data that can be transferred from a single input/output operation to one buffer. The page size is a permanent attribute of the data set and is used when the data set is processed.

A larger page size can improve execution time by reducing the number of times SAS has to read from or write to the storage medium. However, the improvement in execution time comes at the expense of increased memory consumption.

To change the page size, use a DATA step to copy the data set and either specify a new page or use the SAS default.

*Note:* If you use the COPY procedure to copy a data set to another library that is allocated with a different engine, the specified page size of the data set is not retained.

Operating Environment Information: The default value for BUFSIZE= is determined by your operating environment and is set to optimize sequential access. To improve performance for direct (random) access, you should change the value for BUFSIZE=.

For the default setting and possible settings for direct access, see the BUFSIZE= system option in the SAS documentation for your operating environment.

Operating Environment Information: The syntax that is shown here applies to the OPTIONS statement. On the command line or in a configuration file, the syntax is specific to your operating environment. For details, see the SAS documentation for your operating environment.

Comparisons

The BUFSIZE= system option can be overridden by the BUFSIZE= data set option.
BYERR System Option

Controls whether SAS generates an error message and sets the error flag when a _NULL_ data set is used in the SORT procedure

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Environment control: Error handling

PROC OPTIONS GROUP=ERRORHANDLING

Syntax

BYERR | NOBYERR

Syntax Description

BYERR
specifies that SAS issue an error message and stop processing if the SORT procedure attempts to sort a _NULL_ data set or if no BY statement or BY variable is specified.

NOBYERR
specifies that SAS ignore the error message and continue processing if the SORT procedure attempts to sort a _NULL_ data set or if no BY statement or BY variable is specified.

Comparisons

The VNFERR system option sets the error flag for a missing variable when a _NULL_ data set is used. The DSNFERR system option controls how SAS responds when a SAS data set is not found.

See Also

System Options:
“DSNFERR System Option” on page 1517
“VNFERR System Option” on page 1627
“BY-Group Processing” in SAS Language Reference: Concepts
BYLINE System Option

Controls whether BY lines are printed above each BY group

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Log and procedure output control: Procedure output

PROC OPTIONS GROUP= LISTCONTROL

Syntax

BYLINE | NOBYLINE

Syntax Description

BYLINE
specifies that BY lines are printed above each BY group.

NOBYLINE
suppresses the automatic printing of BY lines.

Details

Use NOBYLINE to suppress the automatic printing of BY lines in procedure output. You can then use #BYVAL, #BYVAR, or #BYLINE to display BYLINE information in a TITLE statement.

These SAS procedures perform their own BY-line processing by displaying output for multiple BY groups on the same page:

- MEANS
- PRINT
- STANDARD
- SUMMARY
- TTEST (in SAS/STAT software).

With these procedures, NOBYLINE causes a page eject between BY groups. For PROC PRINT, the page eject between BY groups has the same effect as specifying the right most BY variable on the PAGEBY statement.

See Also

Statements:

#BYVAL, #BYVAR, and #BYLINE in the “TITLE Statement” on page 1409

“BY-Group Processing in SAS Programs” in SAS Language Reference: Concepts
**BYSORTED System Option**

Specifies whether observations in one or more data sets are sorted in alphabetic or numeric order or are grouped in another logical order.

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Input control: Data Processing

**PROC OPTIONS GROUP= INPUTCONTROL**

**Syntax**

BYSORTED | NOBYSORTED

**Syntax Description**

**BYSORTED**

specifies that observations in a data set or data sets are sorted in alphabetic or numeric order.

**CAUTION:**
If the BYSORTED system option and the NOTSORTED statement option on a BY statement are both specified, then the NOTSORTED option in the BY statement takes precedence over the BYSORTED system option. △

**Requirement:** When you use the BYSORTED option, observations must be ordered or indexed according to the values of BY variables.

**Tip:** If BYSORTED is specified, then SAS assumes that the data set is ordered by the BY variable. BYSORTED should be used if the data set is ordered by the BY variable for better performance.

**NOBYSORTED**

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

**Tip:** When the NOBYSORTED option is specified, you do not have to specify NOTSORTED on every BY statement to access the data set(s).

**Tip:** NOBYSORTED is useful if you have data that falls into other logical groupings such as chronological order or linguistic order. This allows BY processing to continue without failure when a data set is not actually sorted in alphabetic or numeric order.

**Note:** If a procedure ignores the NOTSORTED option in a BY statement, then it ignores the NOBYSORTED system option also. △

**Details**

The requirement for ordering or indexing observations according to the values of BY variables is suspended for BY-group processing when you use the NOBYSORTED option. By default, BY-group processing requires that your data be sorted in alphabetic or numeric order. If your data is grouped in any other way but alphabetic or numeric, then you must use the NOBYSORTED option to allow BY-processing to continue.
without failure. For more information on BY-group processing, see “BY-Group Processing in SAS Programs” in SAS Language Reference: Concepts.

See Also

Statements:

NOTSORTED option in the “BY Statement” on page 1114.

CAPS System Option

Indicates whether to translate input to uppercase

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Input control: Data Processing

PROC OPTIONS GROUP= INPUTCONTROL

Syntax

CAPS | NOCAPS

Syntax Description

CAPS

specifies that SAS translate lowercase characters to uppercase in these types of input:

- data following CARDS, CARDS4, DATALINES, DATALINES4, and PARMCARDS statements
- text enclosed in single or double quotation marks
- values in VALUE and INVALUE statements in the FORMAT procedure
- titles, footnotes, variable labels, and data set labels
- constant text in macro definitions
- values of macro variables
- parameter values passed to macros.

Note: Data read from external files and SAS data sets are not translated to uppercase. △

NOCAPS

specifies that lowercase characters that occur in the types of input that are listed above are not translated to uppercase.

Comparisons

The CAPS system option and the CAPS command both specify whether input is converted to uppercase. The CAPS command, which is available in windows that allow text editing, can act as a toggle. The CAPS command converts all text that is entered
from the keyboard to uppercase. If either the CAPS system option or the CAPS command is in effect, all applicable input is translated to uppercase.

See Also

Command:
CAPS in SAS Help and Documentation

CARDIMAGE System Option

Processes SAS source and data lines as 80-byte cards

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Input control: Data Processing

PROC OPTIONS GROUP= INPUTCONTROL

See: CARDIMAGE System Option in the documentation for your operating environment.

Syntax

CARDIMAGE | NOCARDIMAGE

Syntax Description

CARDIMAGE
specifies that SAS source and data lines be processed as if they were punched card images—all exactly 80 bytes long and padded with blanks. That is, column 1 of a line is treated as if it immediately followed column 80 of the previous line. Therefore, tokens can be split across lines. (A token is a character or series of characters that SAS treats as a discrete word.)

Strings in quotation marks (literal tokens) that begin on one line and end on another are treated as if they contained blanks out to column 80 of the first line. Data lines longer than 80 bytes are split into two or more 80-byte lines. Data lines are not truncated regardless of their length.

NOCARDIMAGE

specifies that SAS source and data lines not be treated as if they were 80-byte card images. When NOCARDIMAGE is in effect, the end of a line is always treated as the end of the last token, except for strings in quotation marks. Strings in quotation marks can be split across lines. Other types of tokens cannot be split across lines under any circumstances. Strings in quotation marks that are split across lines are not padded with blanks.

Operating Environment Information: CARDIMAGE is generally used in the z/OS operating environment; NOCARDIMAGE is used in other operating environments.
Examples

Consider the following DATA step:

```sas
data;
  x='A B';
run;
```

If CARDIMAGE is in effect, the variable X receives a value that consists of 78 characters: the A, 76 blanks, and the B. If NOCARDIMAGE is in effect, the variable X receives a value that consists of two characters: AB, with no intervening blanks.

---

**CATCACHE= System Option**

Specifies the number of SAS catalogs to keep open

Valid in: configuration file, SAS invocation

Category: Files: SAS Files

PROC OPTIONS GROUP= SASFILES

See: CATCACHE= System Option in the documentation for your operating environment.

---

**Syntax**

```
CATCACHE=n | hex | MIN | MAX |
```

**Syntax Description**

- **n**
  - specifies any integer greater than or equal to 0 in terms of bytes. If \( n > 0 \), SAS places up to that number of open-file descriptors in cache memory instead of closing the catalogs.

- **hex**
  - specifies the number of open-file descriptors that are kept in cache memory as a hexadecimal number.

  You must specify the value beginning with a number (0-9), followed by an X. For example, the value **2dx** sets the number of catalogs to keep open to 45 catalogs.

- **MIN**
  - sets the number of open-file descriptors that are kept in cache memory to 0.

- **MAX**
  - sets the number of open-file descriptors that are kept in cache memory to the largest, signed, 4–byte integer representable in your operating environment.

**CAUTION:**

- The recommended maximum setting for this option is **10**.
Details
Use the CATCACHE= system option to tune an application by avoiding the overhead of repeatedly opening and closing the same SAS catalogs.

*Operating Environment Information:* The syntax that is shown here applies to the OPTIONS statement. On the command line or in a configuration file, the syntax is specific to your operating environment. For details, see the SAS documentation for your operating environment.

*Operating Environment Information:* Some system settings may affect the default setting. See the documentation for your operating system for more information.

---

**CBUFNO= System Option**

Controls the number of extra page buffers to allocate for each open SAS catalog

**Valid in:** configuration file, SAS invocation, OPTIONS statement, SAS System Options window

**Category:** Files: SAS Files

**PROC OPTIONS GROUP=** SASFILES

---

**Syntax**

CBUFNO=n | nK | nM | nG | nT | hexX | MIN | MAX

**Syntax Description**

- **n | nK | nM | nG | nT**
  - specifies the number of extra page buffers in multiples of 1 (bytes); 1,024 (kilobytes); 1,048,576 (megabytes); 1,073,741,824 (gigabytes); or 1,099,511,627,776 (terabytes).
  - For example, a value of 8 specifies 8 bytes, and a value of 3m specifies 3,145,728 bytes.

- **MIN**
  - sets the number of extra page buffers to 0.

- **MAX**
  - sets the number of extra page buffers to 20.

- **hexX**
  - specifies the number of extra page buffers as a hexadecimal number. You must specify the value beginning with a number (0–9), followed by an X.
  - For example, the value 0ax sets the number of extra page buffers to 10 buffers.

**Details**
The CBUFNO= option is similar to the BUFNO= option that is used for SAS data set processing.

Increasing the value for the CBUFNO= option might result in fewer I/O operations when your application reads very large objects from catalogs. Increasing this value also comes with the normal tradeoff between performance and memory usage. If memory is a serious constraint for your system, you probably should not increase the value of the
CBUFNO= option. This is especially true if you have increased the value of the CATCACHE= option.

Operating Environment Information: The syntax that is shown here applies to the OPTIONS statement. On the command line or in a configuration file, the syntax is specific to your operating environment. For details, see the SAS documentation for your operating environment.

---

**CENTER System Option**

Controls alignment of SAS procedure output

**Valid in:** configuration file, SAS invocation, OPTIONS statement, SAS System Options window

**Category:** Log and procedure output control: Procedure output

**PROC OPTIONS GROUP=** LISTCONTROL

---

**Syntax**

CENTER | NOCENTER

---

**Syntax Description**

CENTER
centers SAS procedure output.

NOCENTER
left aligns SAS procedure output.

---

**CHARCODE System Option**

Determines whether character combinations are substituted for special characters that are not on the keyboard

**Valid in:** configuration file, SAS invocation, OPTIONS statement, SAS System Options window

**Category:** Environment control: Display

**PROC OPTIONS GROUP=** ENVDISPLAY

---

**Syntax**

CHARCODE | NOCHARCODE
Syntax Description

CHARCODE
allows certain character combinations to be substituted for special characters that may not be on your keyboard.

NOCHARCODE
does not allow substitutions for certain keyboard characters.

Details
If you do not have the following symbols on your keyboard, you can use these character combinations to create the symbols that you need when CHARCODE is active:

<table>
<thead>
<tr>
<th>To create:</th>
<th>Use:</th>
</tr>
</thead>
<tbody>
<tr>
<td>backquote (')</td>
<td>?</td>
</tr>
<tr>
<td>backslash ()</td>
<td>,</td>
</tr>
<tr>
<td>left brace ({)</td>
<td>?(</td>
</tr>
<tr>
<td>right brace (})</td>
<td>?)</td>
</tr>
<tr>
<td>logical not sign (¬, or ^)</td>
<td>?=</td>
</tr>
<tr>
<td>left square bracket ([)</td>
<td>?&lt;</td>
</tr>
<tr>
<td>right square bracket (])</td>
<td>?&gt;</td>
</tr>
<tr>
<td>underscore (_)</td>
<td>?-</td>
</tr>
<tr>
<td>vertical bar (</td>
<td>)</td>
</tr>
</tbody>
</table>

Examples
This statement produces the output [TEST TITLE]:

title '??<TEST TITLE??';

CLEANUP System Option

Specifies how to handle an out-of-resource condition
Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window
Category: Environment control: Error handling
PROC OPTIONS GROUP= ERRORHANDLING
See: CLEANUP System Option in the documentation for your operating environment.

Syntax
CLEANUP | NOCLEANUP
Syntax Description

CLEANUP

specifies that during the entire session, SAS attempts to perform automatic,
continuous clean-up of resources that are not essential for execution. Nonessential
resources include those that are not visible to the user (for example, cache memory)
and those that are visible to the user (for example, the KEYS windows).

When CLEANUP is in effect and an out-of-resource condition occurs (except for a
disk-full condition), a requester window is not displayed, and no intervention is
required by the user. When CLEANUP is in effect and a disk-full condition occurs, a
requester window displays that allows the user to decide how to proceed.

NOCLEANUP

specifies that SAS allow the user to choose how to handle an out-of-resource
condition. When NOCLEANUP is in effect and SAS cannot execute because of a lack
of resources, SAS automatically attempts to clean up resources that are not visible to
the user (for example, cache memory). However, resources that are visible to the user
(for example, windows) are not automatically cleaned up. Instead, a requester
window appears that allows the user to choose how to proceed.

Details

This table lists the requester window choices:

<table>
<thead>
<tr>
<th>Requester Window Choice</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free windows</td>
<td>clears all windows not essential for execution.</td>
</tr>
<tr>
<td>Clear paste buffers</td>
<td>deletes paste buffer contents.</td>
</tr>
<tr>
<td>Deassign inactive librefs</td>
<td>prompts user for librefs to delete.</td>
</tr>
<tr>
<td>Delete definitions of all SAS macros and macro variables</td>
<td>deletes all macro definitions and variables.</td>
</tr>
<tr>
<td>Delete SAS files</td>
<td>allows user to select files to delete.</td>
</tr>
<tr>
<td>Clear Log window</td>
<td>erases Log window contents.</td>
</tr>
<tr>
<td>Clear Output window</td>
<td>erases Output window contents.</td>
</tr>
<tr>
<td>Clear Program Editor window</td>
<td>erases Program Editor window contents.</td>
</tr>
<tr>
<td>Clear source spooling/DMS recall buffers</td>
<td>erases recall buffers.</td>
</tr>
<tr>
<td>More items to clean up</td>
<td>displays a list of other resources that can be cleaned up.</td>
</tr>
</tbody>
</table>
### Requester Window Choice

<table>
<thead>
<tr>
<th>Choice</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clean up everything</td>
<td>cleans up all other options that are shown on the requestor window. This selection only applies to the current clean-up request, not to the entire SAS session.</td>
</tr>
<tr>
<td>Continuous clean up</td>
<td>performs automatic, continuous clean-up. When continuous clean up is selected, SAS cleans up as many resources as possible in order to continue execution, and it ceases to display the requester window. Selecting continuous clean-up has the same effect as specifying CLEANUP. This selection applies to the current clean-up request and to the remainder of the SAS session.</td>
</tr>
</tbody>
</table>

### Operating Environment Information:

Some operating environments may also include these choices in the requester window:

<table>
<thead>
<tr>
<th>Requester Window Choice</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute X command</td>
<td>enables the user to erase files and perform other clean-up operations.</td>
</tr>
<tr>
<td>Do nothing</td>
<td>halts the clean-up request and returns to the SAS session. This selection only applies to the current clean-up request, not to the entire SAS session.</td>
</tr>
</tbody>
</table>

If an out-of-resource condition cannot be resolved, the requester window continues to display. In that case, see the SAS documentation for your operating environment for instructions on terminating the SAS session.

When running in modes other than a windowing environment, the operation of CLEANUP depends on your operating environment. For details, see the SAS documentation for your operating environment.

### CMDMAC System Option

**Determines whether the macro processor recognizes a command-style macro invocation**

**Valid in:** configuration file, SAS invocation, OPTIONS statement, SAS System Options window

**Category:** Macro: SAS macro

**PROC OPTIONS GROUP:** MACRO

**See:** The CMDMAC System Option in *SAS Macro Language: Reference*. 
CMPLIB= System Option

Specifies one or more SAS catalogs containing compiler subroutines to include during program compilation

Valid in: configuration file, SAS invocation, OPTIONS statement, System Options window
Category: Files: SAS Files
PROC OPTIONS GROUP= SASFILES

Syntax

CMPLIB=libref.catalog | (libref.catalog-1 ... libref.catalog-n) | (libref.catalogN - libref.catalogM)

Syntax Description

libref.catalog
specifies the libref and the catalog of the compiler subroutines that are to be included during program compilation. The libref and catalog must be valid SAS names.

libref.catalogN - libref.catalogM
specifies a range of compiler subroutines that are to be included during program compilation. The name of the libref and the catalog must be valid SAS names containing a numeric suffix.

Details

SAS procedures that perform non-linear statistical modeling or optimization employ a SAS language compiler subsystem that compiles and executes your SAS programs. The compiler subsystem generates machine language code for the computer on which SAS is running. The SAS procedures that use the SAS language compiler are CALIS, COMPARE, GENMOD, MODEL, PHREG, NLIN, NLMIXED, NLP, RISK, and SYLK.

The subroutines that you want to include must already have been compiled. All the subroutines in libref.catalog are included.

You can specify a single libref.catalog, a list of libref.catalog names, or a range of libref.catalog names with numeric suffixes. When you specify more than one libref.catalog name, separate the names with a space and enclose the names in parentheses.

Examples

<table>
<thead>
<tr>
<th>Number of Libraries</th>
<th>OPTIONS Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>One library</td>
<td>options cmplib=sasuser.cmpl;</td>
</tr>
<tr>
<td>Two or more libraries</td>
<td>options cmplib=(sasuser.cmpl sasuser.cmplA sasuser.cmpl3);</td>
</tr>
<tr>
<td>A range of libraries</td>
<td>options cmplib=(sasuser.cmpl1 - sasuser.cmpl6);</td>
</tr>
</tbody>
</table>
CMPOPT= System Option

Specifies the type of code generation optimizations to use in the SAS language compiler

Valid in: configuration file, SAS invocation, OPTIONS statement, System Options window

Category: System administration: Performance

PROC OPTIONS GROUP= PERFORMANCE

Syntax

CMPOPT=optimization-value | (optimization-value-1 ... optimization-value-n) | "optimization-value-1 ... optimization-value-n" | ALL | NONE

NOCMPOPT

CMPOPT Syntax Description

optimization

specifies the type of optimization that the SAS compiler is to use. Valid values are

EXTRAMATH | NOEXTRAMATH

specifies to keep or remove mathematical operations that do not affect the outcome of a statement. When you specify EXTRAMATH, the compiler retains the extra mathematical operations. When you specify NOEXTRAMATH, the extra mathematical operations are removed.

MISSCHECK | NOMISSCHECK

specifies whether to check for missing values in the data. If the data contains a significant amount of missing data, then you can optimize the compilation by specifying MISSCHECK. If the data rarely contains missing values, then you can optimize the compilation by specifying NOMISSCHECK.

PRECISE | NOPRECISE

specifies to handle exceptions at an operation boundary or at a statement boundary. When you specify PRECISE, exceptions are handled at the operation boundary. When you specify NOPRECISE, exceptions are handled at the statement boundary.

GUARDCHECK | NOGUARDCHECK

specifies whether to check for array boundary problems. When you specify GUARDCHECK, the compiler checks for array boundary problems. When you specify NOGUARDCHECK, the compiler does not check for array boundary problems.

Note: NOGUARDCHECK is set when CMPOPT is set to ALL and when CMPOPT is set to NONE.

Tip: EXTRAMATH, MISSCHECK, PRECISE, and GUARDCHECK can be specified in any combination when you specify one or more values.

ALL

specifies that the compiler is to optimize the machine language code by using the (NOEXTRAMATH NOMISSCHECK NOPRECISE NOGUARDCHECK) optimization values. ALL cannot be specified in combination with any other values. This is the default.
NONE
specifies that the compiler is not set to optimize the machine language code by using the (EXTRAMATH MISSCHECK PRECISE NOGUARDCHECK) optimization values. NONE cannot be specified in combination with any other values.

NOCMPOPT Syntax Description

NOCMPOPT
specifies to set the value of CMPOPT to ALL. The compiler is to optimize the machine language code by using the (NOEXTRAMATH NOMISSCHECK NOPRECISE NOGUARDCHECK) optimization values.

Restriction: NOCMPOPT cannot be specified in combination with values for the CMPOPT option.

Details
SAS procedures that perform non-linear statistical modeling or optimization employ a SAS language compiler subsystem that compiles and executes your SAS programs. The compiler subsystem generates machine language code for the computer on which SAS is running. By specifying values with the CMPOPT option, the machine language code can be optimized for efficient execution. The SAS procedures that use the SAS language compiler are CALIS, COMPILE, GENMOD, MODEL, PHREG, NLIN, NLMIXED, NLP, RISK, and SYLK.

To specify multiple optimization values, the values must be enclosed in either parentheses, single quotation marks, or double quotation marks. When CMPOPT is set to multiple values, the parentheses or quotation marks are retained as part of the value. They are not retained as part of the value when CMPOPT is set to a single value.

If a value is entered more than once, then the last setting is used. For example, if you specify CMPOPT=(PRECISE NOEXTRAMATH NOPRECISE), then the values that are set are NOEXTRAMATH and NOPRECISE. All leading, trailing, and embedded blanks are removed.

When you specify EXTRAMATH or NOEXTRAMATH, some of the mathematical operations that are either included or excluded in the machine language code are

\[
\begin{align*}
  x & \times 1 \\
  x & \div 1 \\
  x & + 0 \\
  x & - x \\
  -x & \\
\end{align*}
\]

any operation on two literal constants
Collate System Option

Specifies the collation of multiple copies for output to a printer

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Log and procedure output control: ODS printing

PROC OPTIONS GROUP= ODSPRINT

Syntax

COLLATE | NOCOLLATE

Syntax Description

COLLATE
specifies collating multiple copies of output.

NOCOLLATE
specifies not collating multiple copies of output. This is the default.

Details

When you send a print job to the printer and you want multiple copies of multiple pages, the COLLATE option controls how the pages are ordered:

- COLLATE causes the pages to print consecutively: 123, 123, 123...
- NOCOLLATE causes the same-numbered pages to print together: 111, 222, 333...

Note: You can also control collation with the SAS windowing environment Page Setup window, invoked with the DMPAGESETUP command.

Most SAS system options are initialized with default settings when SAS is invoked. However, the default settings and/or option values for some SAS system options may vary both by operating environment and by site. For details, see the SAS documentation for your operating environment.

Examples

<table>
<thead>
<tr>
<th>When the OPTIONS statement is...</th>
<th>The result is...</th>
</tr>
</thead>
<tbody>
<tr>
<td>options cmpopt=(extramath);</td>
<td>extramath</td>
</tr>
<tr>
<td>options cmpopt=&quot;extramath misscheck precise&quot;;</td>
<td>&quot;precise extramath extramath&quot;</td>
</tr>
<tr>
<td>options nocmpopt;</td>
<td>(noextramath nomisscheck noprecise noguardcheck)</td>
</tr>
</tbody>
</table>
For additional information on declaring an ODS printer destination, see ODS statements in SAS Output Delivery System: User’s Guide.
For additional information on the SAS universal print facility, see “Printing with SAS” in SAS Language Reference: Concepts.

See Also

System Option:
“COPIES= System Option” on page 1502

COLORPRINTING System Option

Specifies color printing to a printer, if color printing is supported

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window
Category: Log and procedure output control: ODS printing
PROC OPTIONS GROUP= ODSPRINT

Syntax
COLORPRINTING | NOCOLORPRINTING

Syntax Description

COLORPRINTING
specifies to attempt to print in color.

NOCOLORPRINTING
specifies not to print in color.

Details
Most SAS system options are initialized with default settings when SAS is invoked. However, the default settings and/or option values for some SAS system options may vary both by operating environment and by site. For details, see the SAS documentation for your operating environment.
For additional information on declaring an ODS printer destination, see ODS statements in SAS Output Delivery System: User’s Guide.
For additional information on the SAS universal print facility, see “Printing with SAS” in SAS Language Reference: Concepts.

COMPRESS= System Option

Controls the compression of observations in output SAS data sets
Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Files: SAS Files

PROC OPTIONS GROUP= SASFILES

Restriction: The TAPE engine does not support the COMPRESS= system option.

Syntax

COMPRESS=NO | YES | CHAR | BINARY

Syntax Description

**NO**

specifies that the observations in a newly created SAS data set are uncompressed (fixed-length records).

*Alias:* N

**YES | CHAR**

specifies that the observations in a newly created SAS data set are compressed (variable-length records) by SAS using RLE (Run Length Encoding). RLE compresses observations by reducing repeated consecutive characters (including blanks) to two-byte or three-byte representations.

*Alias:* Y, ON

*Tip:* Use this compression algorithm for character data.

*Note:* COMPRESS=CHAR is accepted by Version 7 and later versions.

**BINARY**

specifies that the observations in a newly created SAS data set are compressed (variable-length records) by SAS using RDC (Ross Data Compression). RDC combines run-length encoding and sliding-window compression to compress the file.

*Tip:* This method is highly effective for compressing medium to large (several hundred bytes or larger) blocks of binary data (numeric variables). Because the compression function operates on a single record at a time, the record length needs to be several hundred bytes or larger for effective compression.

Operating Environment Information: The syntax that is shown here applies to the OPTIONS statement. On the command line or in a configuration file, the syntax is specific to your operating environment. For details, see the SAS documentation for your operating environment.

Details

Compressing a file is a process that reduces the number of bytes required to represent each observation. Advantages of compressing a file include reduced storage requirements for the file and fewer I/O operations necessary to read or write to the data during processing. However, more CPU resources are required to read a compressed file (because of the overhead of uncompressing each observation), and there are situations when the resulting file size may increase rather than decrease.

Use the COMPRESS= system option to compress all output data sets that are created during a SAS session. Use the option only when you are creating SAS data files (member type DATA). You cannot compress SAS views, because they contain no data.
Once a file is compressed, the setting is a permanent attribute of the file, which means that to change the setting, you must re-create the file. That is, to uncompress a file, specify COMPRESS=NO for a DATA step that copies the compressed file.

**Comparisons**

The COMPRESS= system option can be overridden by the COMPRESS= option on the LIBNAME statement and the COMPRESS= data set option.

The data set option POINTOBS=YES, which is the default, determines that a compressed data set can be processed with random access (by observation number) rather than sequential access. With random access, you can specify an observation number in the FSEDIT procedure and the POINT= option in the SET and MODIFY statements.

When you create a compressed file, you can also specify REUSE=YES (as a data set option or system option) in order to track and reuse space. With REUSE=YES, new observations are inserted in space freed when other observations are updated or deleted. When the default REUSE=NO is in effect, new observations are appended to the existing file.

POINTOBS=YES and REUSE=YES are mutually exclusive; that is, they cannot be used together. REUSE=YES takes precedence over POINTOBS=YES; that is, if you set REUSE=YES, SAS automatically sets POINTOBS=NO.

The TAPE engine does not support the COMPRESS= system option, but the engine does support the COMPRESS= data set option.

The XPORT engine does not support compression.

**See Also**

Data Set Options:
- “COMPRESS= Data Set Option” on page 14
- “POINTOBS= Data Set Option” on page 42
- “REUSE= Data Set Option” on page 49

Statements:
- “LIBNAME Statement” on page 1286

System Option:
- “REUSE= System Option” on page 1591
- “Compressing Data Files” in *SAS Language Reference: Concepts*

---

**COPIES= System Option**

**Specifies the number of copies to make when printing to a printer**

**Valid in:** configuration file, SAS invocation, OPTIONS statement, SAS System Options window

**Category:** Log and procedure output control: ODS printing

**PROC OPTIONS GROUP=** ODSPRINT

**Syntax**

COPIES=n
Syntax Description

\( n \)

specifies the number of copies.

Operating Environment Information: Most SAS system options are initialized with default settings when SAS is invoked. However, the default settings and/or option values for some SAS system options may vary both by operating environment and by site. For details, see the SAS documentation for your operating environment. \( \triangle \)

For additional information on declaring an ODS printer destination, see ODS statements in SAS Output Delivery System: User’s Guide. For additional information on the SAS universal print facility, see “Printing with SAS” in SAS Language Reference: Concepts.

See Also

System Option:

“COLLATE System Option” on page 1499

### CPUCOUNT= System Option

Specifies the number of processors that the thread-enabled applications should assume will be available for concurrent processing

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: System administration: Performance

PROC OPTIONS GROUP= PERFORMANCE

Interaction: If the THREADS system option is set to NOTHREADS, the CPUCOUNT= option has no effect.

Syntax

CPUCOUNT= 1 - 1024 | ACTUAL

Syntax Description

1-1024

is the number of CPUs that SAS will assume are available for use by thread-enabled applications.

**CAUTION:**

Setting CPUCOUNT= to a number greater than the actual number of available CPUs might result in reduced overall performance of SAS. \( \triangle \)

Tip: This is typically set to the actual number of CPUs available to the current process by your configuration.
ACTUAL
returns the number of physical processors available when the option is set.

Tip: This number can be less than the physical number of CPUs if the SAS process has been restricted by system administration tools.

Tip: Setting CPUCOUNT= to ACTUAL at any time causes the option to be reset to the actual CPU count as seen by the SAS session at that time.

Details
In SAS 9 and SAS 9.1, certain procedures have been modified to take advantage of multiple CPUs by threading the procedure processing. The Base SAS engine also uses threading to create an index. The CPUCOUNT= option provides the information that is needed to make decisions about the allocation of threads.

Changing the value of CPUCOUNT= affects the degree of parallelism each thread-enabled process attempts to achieve. Setting CPUCOUNT to a number greater than the actual number of available CPUs might result in reduced overall performance of SAS.

Comparisons
When the related system option THREADS is in effect, threading will be active where available. The value of the CPUCOUNT= option affects the performance of THREADS by suggesting how many system CPUs are available for use by thread-enabled SAS procedures.

See Also

System Options:
“THREADS System Option” on page 1615
“UTILLOC= System Option” on page 1620

CPUID System Option

Specifies whether hardware information is written to the SAS log
Valid in: configuration file, SAS invocation
Category: Log and procedure output control: SAS log
PROC OPTIONS GROUP= LOGCONTROL

Syntax
CPUID | NOCPUID
Syntax Description

CPUID
specifies that a note that contains the CPU identification number is printed at the top of the SAS log after the licensing information.

NOCPUID
specifies that the note that contains the CPU identification number is not written to the SAS log.

DATASTMTCHK= System Option

Prevents certain errors by controlling the SAS keywords that are allowed in the DATA statement

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window
Category: Files: SAS Files
PROC OPTIONS GROUP= SASFILES

Syntax
DATASTMTCHK=COREKEYWORDS | ALLKEYWORDS | NONE

Syntax Description

COREKEYWORDS
(default) prohibits certain words as one-level SAS data set names in the DATA statement. They can appear as two-level names, however. When you specify COREKEYWORDS, the keywords that cannot appear as one-level SAS data set names are
  MERGE
  RETAIN
  SET
  UPDATE.

For example, SET is not acceptable in the DATA statement, but SAVE.SET and WORK.SET are acceptable.

ALLKEYWORDS
prohibits any keyword that can begin a statement in the DATA step (for example, ABORT, ARRAY, INFILE) as a one-level data set name in the DATA statement.

NONE
provides no protection against overwriting SAS data sets.

Details
It is possible to wipe out unintentionally an input data set when you omit a semicolon on the DATA statement. If the next statement is SET, MERGE, or UPDATE, the
original data set is overwritten. Different, but significant, problems arise when the next statement is RETAIN. DATASTMTCHK= enables you to protect yourself against overwriting the input data set.

**DATE System Option**

Prints the date and time that the SAS session was initialized

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Log and procedure output control: SAS log and procedure output

PROC OPTIONS GROUP= LOG_LISTCONTROL

**Syntax**

DATE | NODATE

**Syntax Description**

DATE

specifies that the date and the time that the SAS job began are printed at the top of each page of the SAS log and any output that is created by SAS.

Note: If the SAS session is in interactive mode, the date/time is not noted in the log window. It is noted instead in the output window.

NODATE

specifies that the date and the time are not printed.

**DATESTYLE= System Option**

Identifies the sequence of month, day, and year when ANYDTDTE, ANYDTDTM, or ANYDTTME informat data is ambiguous

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Environment control: Language control

Input control: Data Processing

PROC OPTIONS GROUP= INPUTCONTROL LANGUAGECONTROL

**Syntax**

DATESTYLE= MDY | MYD | YMD | YDM | DMY | DYM | LOCALE
Syntax Description

MDY
specifies that SAS set the order as month, day, year.

MYD
specifies that SAS set the order as month, year, day.

YMD
specifies that SAS set the order as year, month, day.

YDM
specifies that SAS set the order as year, day, month.

DMY
specifies that SAS set the order as day, month, year.

DYM
specifies that SAS set the order as day, year, month.

LOCALE
specifies that SAS set the order based on the value that corresponds to the LOCALE= system option value and is one of the following: MDY | MYD | YMD | YDM | DMY | DYM.

Details

System option DATESTYLE= identifies the order of month, day, and year. The default value is LOCALE. The default LOCALE system option value is English, therefore, the default DATESTYLE order is MDY.


See Also

System Option:

Informat:
“ANYYMDTDEw. Informat” on page 968
“ANYYMDTDMw. Informat” on page 970
“ANYYMTTMEw. Informat” on page 971

DETAILS System Option

Specifies whether to include additional information when files are listed in a SAS data library

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Log and procedure output control: SAS log and procedure output

PROC OPTIONS GROUP= LOG_LISTCONTROL
Syntax
DETAILS | NODETAILS

Syntax Description

DETAILS
includes additional information when some SAS procedures and windows display a listing of files in a SAS data library.

NODETAILS
does not include additional information.

Details
The DETAILS specification sets the default display for these components of SAS:
- the CONTENTS procedure
- the DATASETS procedure.

The type and amount of additional information that displays depends on which procedure or window you use.

DEVICE= System Option

Specifies a terminal device driver for SAS/GRAPH software

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Alias: DEV=

Category: Graphics: Driver settings

PROC OPTIONS GROUP= GRAPHICS

See: DEVICE= System Option in the documentation for your operating environment.

Syntax
DEVICE=device-driver-specification

Syntax Description

device-driver-specification
specifies the name of a terminal device driver.

Details
If you omit the device-driver name, you are prompted to enter a driver name when you execute a procedure that produces graphics.
Operating Environment Information: Valid device-driver names depend on your operating environment. For a list of valid device-driver names, refer to the SAS documentation for your operating environment.

The syntax that is shown applies to the OPTIONS statement. However, when you specify DEVICE= either on the command-line or in a configuration file, the syntax is specific to your operating environment and may include additional or alternate punctuation.

---

**DFLANG= System Option**

Specifies language for international date informats and formats

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Environment control: Language control

PROC OPTIONS GROUP= LANGUAGECONTROL


---

**DKRICOND= System Option**

Controls the level of error detection for input data sets during processing of DROP=, KEEP=, and RENAME= data set options

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Files: SAS Files

PROC OPTIONS GROUP= SASFILES

Syntax

DKRICOND=ERROR | WARN | WARNING | NOWARN | NOWARNING

Syntax Description

ERROR

sets the error flag and writes error messages if a variable on a KEEP=, DROP=, or RENAME= data set option or statement does not exist in the data set.

WARN|WARNING

writes warning messages only.

NOWARN|NOWARNING

does not write warning messages.
Examples

In the following statements, if the variable X is not in data set B and DKRICO/N=ERROR, SAS sets the error flag to 1 and displays error messages:

```sas
data a;
  set b(drop=x);
run;
```

See Also

System Option:
“DKRICO/N= System Option” on page 1510

DKRICO/N= System Option

Controls the level of error detection for output data sets during the processing of DROP=, KEEP=, and RENAME= data set options and the corresponding DATA step statements.

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Files: SAS Files

PROC OPTIONS GROUP= SASFILES

Syntax

DKRICO/N=ERROR | WARN | WARNING | NOWARN | NOWARNING

Syntax Description

ERROR
sets the error flag and writes error messages if a variable on a KEEP=, DROP=, or RENAME= option or statement does not exist in the data set.

WARN | WARNING
writes warning messages only.

NOWARN | NOWARNING
does not write warning messages.

Examples

In the following statements, if the variable X is not in data set A and DKRICO/N=ERROR, SAS sets the error flag to 1 and displays error messages:

```sas
data a;
  drop x;
run;
```
**See Also**

System Option:

“DKRICOND= System Option” on page 1509

---

**DLDMSGACTION= System Option**

Specifies what type of action to take when a SAS catalog or a SAS data set in a SAS data library is detected as damaged

**Valid in:** configuration file, SAS invocation, OPTIONS statement, SAS System Options window

**Category:** Files: SAS Files

**PROC OPTIONS GROUP=** SASFILES

---

**Syntax**

DLDMSGACTION=FAIL | ABORT | REPAIR | PROMPT

---

**Syntax Description**

**FAIL**

stops the step and issues an error message to the log immediately. This is the default for batch mode.

**ABORT**

terminates the step and issues an error message to the log, and aborts the SAS session.

**REPAIR**

For catalogs, automatically deletes catalog entries for which an error occurs during the repair process. For data sets, automatically repairs and rebuilds indexes and integrity constraints, unless the data set is truncated. You use the REPAIR statement to restore the truncated data set. It issues a warning message to the log. This is the default for interactive mode.

**PROMPT**

displays a requester window that lets you select FAIL, ABORT, or REPAIR processing for the damaged catalog, data set, or library.

---

**DMR System Option**

Controls the ability to invoke a remote SAS session so that you can run SAS/CONNECT software

**Valid in:** configuration file, SAS invocation
Category: Environment control: Initialization and operation

PROC OPTIONS GROUP= EXECMODES

Syntax

DMR | NODMR

Syntax Description

DMR
enables you to invoke a remote SAS session in order to run SAS/CONNECT software.

NODMR
disables you from invoking a remote SAS session.

Details

You normally invoke the remote SAS session from a local session by including DMR with the SAS command in a script that contains a TYPE statement. (A script is a text file that contains statements to establish or terminate the SAS/CONNECT link between the local and the remote SAS sessions.)

The following SAS execution mode invocation option has precedence over this option:

□ OBJECTSERVER

DMR overrides all other SAS execution mode invocation options. See “Order of Precedence” on page 1447 for more information on invocation option precedence.

See Also

DMR information in SAS/CONNECT User’s Guide

DMS System Option

Invokes the SAS windowing environment

Valid in: configuration file, SAS invocation

Category: Environment control: Initialization and operation

PROC OPTIONS GROUP= EXECMODES

Syntax

DMS | NODMS
Syntax Description

**DMS**
invokes the SAS windowing environment and displays the Log, an Editor window, and Output windows.

**NODMS**
invokes an interactive line mode SAS session.

**Details**
When you invoke SAS and you are using a configuration file or the command line to control your system option settings, it is possible to create a situation where some system option settings conflict with other system option settings. The following invocation system options have precedence over the DMS invocation system option:

- DMR
- OBJECTSERVER.

If you specify DMR while using another invocation option of equal precedence to invoke SAS, SAS uses the last option that is specified. See “Order of Precedence” on page 1447 for more information about invocation option precedence.

**See Also**
System Options:
“DMR System Option” on page 1511
“DMSEXP System Option” on page 1513
“EXPLORER System Option” on page 1528

### DMSEXP System Option

**Invokes SAS with the Explorer, Program Editor, Log, Output, and Results windows**

**Valid in:** configuration file, SAS invocation

**Category:** Environment control: Initialization and operation

**PROC OPTIONS GROUP=** EXECMODES

#### Syntax

`DMSEXP | NODMSEXP`

#### Syntax Description

**DMSEXP**
invokes SAS with the Explorer, Program Editor, Log, Output, and Results windows active.
NODMSEXP
invokes SAS with the Program Editor, Log, and Output windows active.

Details
In order to set DMSEXP or NODMSEXP, the DMS option must be set. The following SAS execution mode invocation options have precedence over this option:
- DMR
- OBJECTSERVER.
DMSEXP has the same precedence as all other SAS execution mode invocation options. If you specify DMSEXP with another execution mode invocation option of equal precedence, SAS uses only the last option listed. See “Order of Precedence” on page 1447 for more information on invocation option precedence.

See Also
System Options:
“DMS System Option" on page 1512
“DMR System Option” on page 1511
“EXPLORER System Option” on page 1528

DMSLOGSIZE= System Option

Specifies the maximum number of rows that the SAS windowing environment Log window can display.
Valid in: configuration file, SAS invocation
Category: Environment control: Display
PROC OPTIONS GROUP= ENVDISPLAY

Syntax
DMSLOGSIZE= n | nK | hexX | MIN | MAX

Syntax Description

\( n \ | \ nK \)
specifies the maximum number of rows that can be displayed in the SAS windowing environment Log window in multiples of 1 (n) or 1,024 (nK). For example, a value of 800 specifies 800 rows, and a value of 3K specifies 3,072 rows. Valid values range from 500 to 999999. The default is 99999.

\( \text{hexX} \)
specifies the maximum number of rows that can be displayed in the SAS windowing environment Log window as a hexadecimal value. You must specify the value beginning with a number (0-9), followed by an X. For example, 2ffX specifies 767 rows and 0a00x specifies 2,560 rows.

\( \text{MIN} \)
specifies to set the maximum number of rows that can be displayed in the SAS windowing environment Log window to 500.

**MAX**

specifies to set the maximum number of rows that can be displayed in the SAS windowing environment Log window to 999999.

**Details**

When the maximum number of rows have been displayed in the Log window, SAS prompts you to either file, print, save, or clear the Log window.

**See Also**

System Option:

“DMSOUTSIZE= System Option” on page 1515

---

**DMSOUTSIZE= System Option**

Specifies the maximum number of rows that the SAS windowing environment Output window can display

Valid in: configuration file, SAS invocation

Category: Environment control: Display

PROC OPTIONS GROUP= ENVDISPLAY

**Syntax**

DMSOUTSIZE= n | nK | hexX | MIN | MAX

**Syntax Description**

$n$ | $nK$

specifies the maximum number of rows that can be displayed in the SAS windowing environment Output window in multiples of $1$ (n) or $1,024$ (nK). For example, a value of 800 specifies 800 rows, and a value of 3K specifies 3,072 rows. Valid values range from 500 to 999999. The default is 99999.

$\text{hexX}$

specifies the maximum number of rows that can be displayed in the SAS windowing environment Output window as a hexadecimal value. You must specify the value beginning with a number (0-9), followed by an X. For example, $\text{2ffX}$ specifies 767 rows and $\text{0A00X}$ specifies 2,560 rows.

**MIN**

specifies to set the maximum number of rows that can be displayed in the SAS windowing environment Output window to 500.

**MAX**

specifies to set the maximum number of rows that can be displayed in the SAS windowing environment Output window to 999999.
Details

When the maximum number of rows have been displayed in the Output window, SAS prompts you to either file, print, save, or clear the Output window.

See Also

System Option:

"DMSLOGSIZE= System Option" on page 1514

DMSSYNCHK System Option

Enables syntax check mode for multiple steps in the SAS windowing environment

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Environment control: Error handling

PROC OPTIONS GROUP=ERRORHANDLING

Syntax

DMSSYNCHK | NODMSSYNCHK

Syntax Description

DMSSYNCHK
  enables syntax check mode for statements that are submitted within the windowing and Explorer environments.

NODMSSYNCHK
  does not enable syntax check mode for statements that are submitted within the windowing and Explorer environments.

Details

If a syntax or semantic error occurs in a DATA step after the DMSSYNCHK option is set, then SAS enters syntax check mode, which remains in effect from the point where SAS encountered the error to the end of the code that was submitted. After SAS enters syntax mode, all subsequent DATA step statements and PROC step statements are validated.

While in syntax check mode, only limited processing is performed. For a detailed explanation of syntax check mode, see “Syntax Check Mode” in the “Error Processing in SAS” section of SAS Language Reference: Concepts.

CAUTION:
Place the OPTIONS statement that enables DMSSYNCHK before the step for which you want it to take effect. If you place the OPTIONS statement inside a step, then DMSSYNCHK will not take effect until the beginning of the next step. △
If NODMSSYNCHK is in effect, SAS processes the remaining steps even if an error occurs in the previous step.

**Comparisons**

You use the DMSSYNCHK system option to validate syntax in an interactive session by using the SAS windowing environment. You use the SYNTAXCHECK system option to validate syntax in a non-interactive or batch SAS session. You can use the ERRORCHECK= option to specify the syntax check mode for the LIBNAME statement, the FILENAME statement, the %INCLUDE statement, and the LOCK statement in SAS/SHARE.

**See Also**

System options:
- “ERRORCHECK= System Option” on page 1527
- “SYNTAXCHECK System Option” on page 1609
- “Error Processing” in *SAS Language Reference: Concepts*

---

**DSNFERR System Option**

**Controls how SAS responds when a SAS data set is not found**

**Valid in:** configuration file, SAS invocation, OPTIONS statement, SAS System Options window

**Category:** Environment control: Error handling

**PROC OPTIONS GROUP=** ERRORHANDLING

---

**Syntax**

DSNFERR | NODSNFERR

**Syntax Description**

**DSNFERR**

specifies that SAS issue an error message and stop processing if a reference is made to a SAS data set that does not exist.

**NODSNFERR**

specifies that SAS ignore the error message and continue processing if a reference is made to a SAS data set that does not exist. The data set reference is treated as if _NULL_ had been specified.
Comparisons

- DSNFERR is similar to the BYERR system option, which issues an error message and stops processing if the SORT procedure attempts to sort a _NULL_ data set.
- DSNFERR is similar to the VNFERR system option, which sets the error flag for a missing variable when a _NULL_ data set is used.

See Also

System Options:
- “BYERR System Option” on page 1485
- “VNFERR System Option” on page 1627

---

DTRESET System Option

Updates date and time in the SAS log and in the listing file

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Log and procedure output control: SAS log and procedure output

PROC OPTIONS GROUP= LOG_LISTCONTROL

Syntax

DTRESET | NODTRESET

Syntax Description

DTRESET
specifies that SAS update the date and time in the titles of the SAS log and the listing file.

NODTRESET
specifies that SAS not update the date and time in the titles of the SAS log and the listing file.

Details

The DTRESET system option updates the date and time in the titles of the SAS log and the listing file. This update occurs when the page is being written. The smallest time increment that is reflected is minutes.

The DTRESET option is especially helpful in obtaining a more accurate date and time stamp when you run long SAS jobs.

When you use NODTRESET, SAS displays the date and time that the job originally started.
**DUPLEX System Option**

Specifies duplexing controls to a printer

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Log and procedure output control: ODS printing

PROC OPTIONS GROUP= ODSPRINT

**Syntax**

DUPLEX | NODUPLEX

**Syntax Description**

**DUPLEX**

specifies that duplexing is performed.

Interaction: When DUPLEX is selected, the setting of the BINDING=option determines how the paper is oriented before output is printed on the second side.

**NODUPLEX**

specifies that duplexing is not performed. This is the default.

**Details**

Note that duplex printing can be used only on printers that support duplex output.

*Operating Environment Information:* Most SAS system options are initialized with default settings when SAS is invoked. However, the default settings for some SAS system options may vary both by operating environment and by site. Option values may also vary both by operating environment and by site. For details, see the SAS documentation for your operating environment.

For additional information on declaring an ODS printer destination, see ODS statements in *SAS Output Delivery System: User's Guide.*

For additional information on the SAS universal print facility, see “Printing with SAS” in *SAS Language Reference: Concepts.*

**See Also**

System Option:

“BINDING= System Option” on page 1480

---

**ECHOAUTO System Option**

Controls whether autoexec code in an input file is echoed to the log
**Syntax**

`ECHOAUTO | NOECHOAUTO`

**Syntax Description**

**ECHOAUTO**
specifies that SAS source lines that are read from the autoexec file be printed in the SAS log.

**NOECHOAUTO**
specifies that SAS source lines that are read from the autoexec file not be printed in the SAS log, even though they are executed.

**Details**
Regardless of the setting of this option, messages that result from errors in the autoexec files are printed in the SAS log.

---

**EMAILAUTHPROTOCOL= System Option**

Specifies the authentication protocol for SMTP E-mail

**Valid in:** configuration file, SAS invocation

**Category:** Communications: Email

**PROC OPTIONS GROUP=** EMAIL

**Syntax**

`EMAILAUTHPROTOCOL= NONE | LOGIN`

**Syntax Description**

**LOGIN**
specifies that the LOGIN authentication protocol is used.

*Note:* When you specify LOGIN, you might also need to specify EMAILID and EMAILPW. If you omit EMAILID, SAS will look up your user ID and use it. If you omit EMAILPW, no password is used.

**NONE**
specifies that no authentication protocol is used.
Comparisons

For the SMTP access method, use this option in conjunction with the EMAILID=, EMAILPW=, EMAILPORT, and EMAILHOST system options. EMAILID= provides the username, EMAILPW= provides the password, EMAILPORT specifies the port to which the SMTP server is attached, EMAILHOST specifies the SMTP server that supports e-mail access for your site, and EMAILAUTHPROTOCOL= provides the protocol.

See Also

System Options:
- “EMAILHOST System Option” on page 1521
- “EMAILID= System Option” on page 1522
- “EMAILPORT System Option” on page 1523
- “EMAILPW= System Option” on page 1524

EMAILHOST System Option

Specifies the Simple Mail Transfer Protocol (SMTP) server that supports e-mail access for your site

Valid in: configuration file, SAS invocation
Category: Communications: Email
PROC OPTIONS GROUP= EMAIL
See: EMAILHOST System Option in the documentation for your operating environment.

Syntax

EMAILHOST <server>

Syntax Description

server

specifies the domain name of the Simple Mail Transfer Protocol (SMTP) server for your site.

Note: The system administrator for your site will provide this information.

Details

Operating Environment Information: To enable the SMTP interface that SAS provides, you must also specify the EMAILSYS=SMTP system option. For information about EMAILSYS, see the documentation for your operating environment.

Comparisons

For the SMTP access method, use this option in conjunction with the EMAILID=, EMAILAUTHPROTOCOL=, EMAILPORT, and EMAILPW system options. EMAILID=
provides the username, EMAILPW= provides the password, EMAILPORT specifies the port to which the SMTP server is attached, EMAILHOST specifies the SMTP server that supports e-mail access for your site, and EMAILAUTHPROTOCOL= provides the protocol.

See Also

System Option:
“EMAILAUTHPROTOCOL= System Option” on page 1520
“EMAILID= System Option” on page 1522
“EMAILPORT System Option” on page 1523
“EMAILPW= System Option” on page 1524

EMAILID= System Option

Specifies the identity of the individual sending e-mail from within SAS

Valid in: configuration file, SAS invocation
Category: Communications: Email
PROC OPTIONS GROUP= EMAIL

Syntax

EMAILID =loginid | profile | emailaddress

Syntax Description

loginid
specifies the login ID for the user running SAS.
Maximum: The maximum number of characters is 32,000.

profile
see documentation for your e-mail system to determine the profile name.

email-address
specifies the fully qualified e-mail address of the user running SAS.
Requirement: If the value of email-address contains a space, you must enclose it in double quotation marks.
Note: The e-mail address is valid only when SMTP is enabled.

Details

The EMAILID= system option specifies the login ID, profile, or e-mail address to use with your e-mail system.

Comparisons

For the SMTP access method, use this option in conjunction with the EMAILAUTHPROTOCOL=, EMAILPW=, EMAILPORT, and EMAILHOST system
options. EMAILID= provides the username, EMAILPW= provides the password, EMAILPORT specifies the port to which the SMTP server is attached, EMAILHOST specifies the SMTP server that supports e-mail access for your site, and EMAILAUTHPROTOCOL= provides the protocol.

**See Also**

System Options:
- “EMAILAUTHPROTOCOL= System Option” on page 1520
- “EMAILHOST System Option” on page 1521
- “EMAILPORT System Option” on page 1523
- “EMAILPW= System Option” on page 1524

---

**EMAILPORT System Option**

Specifies the port to which the Simple Mail Transfer Protocol (SMTP) server is attached

Valid in: configuration file, SAS invocation
Category: Communications: Email
PROC OPTIONS GROUP= EMAIL

**Syntax**

EMAILPORT <portnumber>

**Syntax Description**

*portnumber*

specifies the port number that is used by the SMTP server that you specified on the EMAILHOST option.

*Note:* The system administrator for your site will provide this information.

**Details**

*Operating Environment Information:* If you use the SMTP protocol that SAS provides, you must also specify the EMAILSYS SMTP system option. For information about EMAILSYS, see the documentation for your operating environment.

**Comparisons**

For the SMTP access method, use this option in conjunction with the EMAILID=, EMAILAUTHPROTOCOL=, EMAILPW=, and EMAILHOST system options. EMAILID= provides the username, EMAILPW= provides the password, EMAILPORT specifies the port to which the SMTP server is attached, EMAILHOST specifies the SMTP server that supports email access for your site, and EMAILAUTHPROTOCOL= provides the protocol.
EMAILPW= System Option

Specifies your e-mail login password
Valid in: configuration file, SAS invocation
Category: Communications: Email
PROC OPTIONS GROUP= EMAIL

Syntax

EMAILPW= "password"

PASSWORD
specifies the login password for your login name.

Restriction: If "password" contains a space, you must enclose the value in double quotation marks.

Details

If you do not specify the EMAILID and EMAILPW system options at invocation (and you are not otherwise logged in to your e-mail system), SAS will prompt you for them when you initiate your e-mail.

Comparisons

For the SMTP access method, use this option in conjunction with the EMAILID=, EMAILAUTHPROTOCOL=, EMAILPORT, and EMAILHOST system options. EMAILID= provides the username, EMAILPW= provides the password, EMAILPORT specifies the port to which the SMTP server is attached, EMAILHOST specifies the SMTP server that supports e-mail access for your site, and EMAILAUTHPROTOCOL= provides the protocol.

See Also

System Options:

“EMAILAUTHPROTOCOL= System Option” on page 1520
“EMAILHOST System Option” on page 1521
ENGINE= System Option

Specifies the default access method for SAS libraries

Valid in: configuration file, SAS invocation

Category: Files: SAS Files

PROC OPTIONS GROUP= SASFILES

See: ENGINE= System Option in the documentation for your operating environment.

Syntax

ENGINE=engine-name

Syntax Description

engine-name

specifies an engine name.

Details

The ENGINE= system option specifies which default engine name is associated with a SAS library. The default engine is used when a SAS library points to an empty directory or a new file. The default engine is also used on directory-based systems, which can store more than one SAS file type within a directory. For example, some operating environments can store SAS files from multiple versions in the same directory.

Operating Environment Information: Valid engine names depend on your operating environment. For details, see the SAS documentation for your operating environment.

See Also

“SAS I/O Engines” in SAS Language Reference: Concepts

ERRORABEND System Option

Specifies how SAS responds to errors

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Alias: ERRABEND | NOERRABEND
Category: Environment control: Error handling
PROC OPTIONS GROUP= ERRORHANDLING

Syntax
ERRORABEND | NOERRORABEND

Syntax Description

ERRORABEND
specifies that SAS abend for most errors (including syntax errors) that would normally cause it to issue an error message, set OBS=0, and go into syntax-check mode (in batch mode only).
Tip: Use the ERRORABEND system option with SAS production programs, which presumably should not encounter any errors. If errors are encountered and ERRORABEND is in effect, SAS brings the errors to your attention immediately by abending. ERRORABEND does not affect how SAS handles notes such as invalid data messages.

NOERRORABEND
specifies that SAS handle errors normally, that is, issue an error message, set OBS=0, and go into syntax-check mode (in batch mode only).

ERRORBYABEND System Option

Specifies how SAS responds to BY-group error conditions
Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window
Category: Environment control: Error handling
PROC OPTIONS GROUP= ERRORHANDLING

Syntax
ERRORBYABEND | NOERRORBYABEND

Syntax Description

ERRORBYABEND
specifies that SAS terminate BY-group error conditions that would normally cause it to issue an error message.

NOERRORBYABEND
specifies that SAS handle BY-group errors normally, that is, by issuing an error message and continuing processing.
Details
If SAS encounters one or more BY-group errors while ERRORBYABEND is in effect, SAS brings the errors to your attention immediately by ending your program. ERRORBYABEND does not affect how SAS handles notes that are written to the SAS log.

Note: Use the ERRORBYABEND system option with SAS production programs that should be error free.

See Also
System Option:
“ERRORABEND System Option” on page 1525

ERRORCHECK= System Option
Controls error handling
Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window
Category: Environment control: Error handling
PROC OPTIONS GROUP= ERRORHANDLING

Syntax
ERRORCHECK=NORMAL | STRICT

Syntax Description
NORMAL
does not place the SAS job into syntax-check mode when an error occurs in a LIBNAME or FILENAME statement, or in a LOCK statement in SAS/SHARE software. In addition, the job or session does not abend when a %INCLUDE statement fails due to a non-existent file.

STRICT
places the SAS job into syntax-check mode when an error occurs in a LIBNAME or FILENAME statement, or in a LOCK statement in SAS/SHARE software. In addition, the job or session abends when a %INCLUDE statement fails due to a non-existent file.

ERRORS= System Option
Controls the maximum number of observations for which complete error messages are printed
Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window
Category: Environment control: Error handling
PROC OPTIONS GROUP= ERRORHANDLING

Syntax
ERRORS=n | nK | nM | nG | nT | MIN | MAX | hexX

Syntax Description

n | nK | nM | nG | nT
specifies the number of observations for which error messages are printed in terms of 1 (n); 1,024 (nK); 1,048,576 (nM); 1,073,741,824 (nG); or 1,099,511,627,776 (nT). For example, a value of 8 specifies 8 observations, and a value of 3M specifies 3,145,728 observations.

MIN
sets the number of observations for which error messages are printed to 0.

MAX
sets the maximum number of observations for which error messages are printed to the largest signed, 4-byte integer representable in your operating environment.

hexX
specifies the maximum number of observations for which error messages are printed as a hexadecimal number. You must specify the value beginning with a number (0–9), followed by an X.
   For example, the value 2dx sets the maximum number of observations for which errors are printed to 45 observations.

Details
If data errors are detected in more than n observations, processing continues, but error messages do not print for the additional errors.

Note: If you set ERRORS=0 and an error occurs, a note displays in the log which states that the limit set by the ERRORS option has been exceeded. △

Operating Environment Information: The syntax that is shown here applies to the OPTIONS statement. On the command line or in a configuration file, the syntax is specific to your operating environment. For details, see the SAS documentation for your operating environment. △

EXPLORER System Option

Controls whether you invoke SAS with only the Explorer window
Valid in: configuration file, SAS invocation
Category: Environment control: Initialization and operation
PROC OPTIONS GROUP= EXEMODES
Syntax
EXPLORER | NOEXPLORER

Syntax Description

EXPLORER
specifies that the SAS session be invoked with only the Explorer window.

NOEXPLORER
specifies that the SAS session be invoked without the Explorer window.

Details
The following SAS execution mode invocation options have precedence over this option:
- DMR
- OBJECTSERVER.

EXPLORER has the same precedence as all other SAS execution mode invocation options. If you specify EXPLORER with another execution mode invocation option of equal precedence, SAS uses only the last option listed. See “Order of Precedence” on page 1447 for more information on invocation option precedence.

See Also
System Options:
“DMS System Option” on page 1512
“DMSEXP System Option” on page 1513

FIRSTOBS= System Option

Specifies which observation or record SAS processes first

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Files: SAS Files

PROC OPTIONS GROUP= SASFILES

Syntax
FIRSTOBS= MIN | MAX | n | nK | nM | nG | nT | hexX
**Syntax Description**

**MIN**
sets the number of the first observation or record to process to 1. This is the default.

**MAX**
sets the number of the first observation to process to the maximum number of observations in the data sets or records in the external file, up to the largest eight-byte, signed integer, which is $2^{31}-1$, or approximately 9.2 quintillion observations.

\[ n \mid nK \mid nM \mid nG \mid nT \]
specifies the number of the first observation or record to process in multiples of 1 (bytes); 1,024 (kilobytes); 1,048,576 (megabytes); 1,073,741,824 (gigabytes); or 1,099,511,627,776 (terabytes). For example, a value of 8 specifies 8 bytes, and a value of 3m specifies 3,145,728 bytes.

**hexX**
specifies the number of the first observation or record to process as a hexadecimal value. You must specify the value beginning with a number (0–9), followed by an X. For example, the value 2dx specifies the 45th observation.

**Details**
The FIRSTOBS= system option is valid for all steps for the duration of your current SAS session or until you change the setting. To affect any single SAS data set, use the FIRSTOBS= data set option.

You can apply FIRSTOBS= processing to WHERE processing. For details, see “Processing a Segment of Data That Is Conditionally Selected” in *SAS Language Reference: Concepts*.

**Operating Environment Information:** The syntax that is shown here applies to the OPTIONS statement. On the command line or in a configuration file, the syntax is specific to your operating environment. For details, see the documentation for your operating environment.

**Comparisons**
- You can override the FIRSTOBS= system option by using the FIRSTOBS= data set option and by using the FIRSTOBS= option as a part of the INFILE statement.
- While the FIRSTOBS= system option specifies a starting point for processing, the OBS= system option specifies an ending point. The two options are often used together to define a range of observations or records to be processed.

**Examples**
If you specify FIRSTOBS=50, SAS processes the 50th observation of the data set first.

This option applies to every input data set that is used in a program or a SAS process. In this example, SAS begins reading at the eleventh observation in the data sets OLD, A, and B:

```sas
options firstobs=11;

data a;
```
Data set OLD has 100 observations, data set A has 90, B has 80, and C has 70. To avoid decreasing the number of observations in successive data sets, use the FIRSTOBS= data set option in the SET statement. You can also reset FIRSTOBS=1 between a DATA step and a PROC step.

See Also

Data Set Option:
“FIRSTOBS= Data Set Option” on page 20

Statement:
“INFILE Statement” on page 1224

System Option:
“OBS= System Option” on page 1572
See Also

System Option:

“FMTSEARCH= System Option” on page 1532

FMTSEARCH= System Option

Controls the order in which format catalogs are searched

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Environment control: Files

PROC OPTIONS GROUP= ENVFILES

See: FMTSEARCH= System Option in the documentation for your operating environment.

Syntax

FMTSEARCH=(catalog-specification-1... catalog-specification-n)

Syntax Description

catalog-specification

searches format catalogs in the order listed, until the desired member is found. The value of libref can be either libref or libref.catalog. If only the libref is given, SAS assumes that FORMATS is the catalog name.

Details

The WORK.FORMATS catalog is always searched first, and the LIBRARY.FORMATS catalog is searched next, unless one of them appears in the FMTSEARCH= list.

If a catalog appears in the FMTSEARCH= list, the catalog is searched in the order in which it appears in the list. If a catalog in the list does not exist, that particular item is ignored and searching continues.

Examples

If you specify FMTSEARCH=(ABC DEF.XYZ GHI), SAS searches for requested formats or informats in this order:

1 WORK.FORMATS
2 LIBRARY.FORMATS
3 ABC.FORMATS
4 DEF.XYZ
5 GHI.FORMATS.
If you specify FMTSEARCH=(ABC WORK LIBRARY) SAS searches in this order:
1  ABC.FORMATS
2  WORK.FORMATS
3  LIBRARY.FORMATS.

Because WORK appears in the FMTSEARCH list, WORK.FORMATS is not automatically searched first.

See Also

System Option:
“FMTERR System Option” on page 1531

---

**FONTSLOC= System Option**

Specifies the location that contains the SAS fonts that are loaded by some Universal Printer drivers

Valid in:  configuration file, SAS invocation
Category:  Environment control: Display
PROC OPTIONS GROUP=  ENVDISPLAY
See:  FONTSLOC= System Option in the documentation for your operating environment

**Syntax**

FONTSLOC= “location”

**Syntax Description**

“location” specifies the location of the SAS fonts that are loaded during the SAS session.
The “location” can be either a fileref or an operating system pathname.

*Note:* If “location” is a fileref, you do not need to enclose the value in quotation marks. △

---

**FORMCHAR= System Option**

Specifies the default output formatting characters

Valid in:  configuration file, SAS invocation, OPTIONS statement, SAS System Options window
Category:  Log and procedure output control: Procedure output
PROC OPTIONS GROUP=  LISTCONTROL
See:  FORMCHAR= System Option in the documentation for your operating environment.

Syntax

FORMCHAR= 'formatting-characters'

Syntax Description

'formatting-characters'

specifies any string or list of strings of characters up to 64 bytes long. If fewer than 64 bytes are specified, the string is padded with blanks on the right.

Details

Formatting characters are used to construct tabular output outlines and dividers for various procedures, such as the CALENDAR, FREQ, and TABULATE procedures. If you omit formatting characters as an option in the procedure, the default specifications given in the FORMCHAR= system option are used. Note that you can also specify a hexadecimal character constant as a formatting character. When you use hex constant with this option, SAS interprets the value of the hexadecimal constant as appropriate for your operating system.

See Also

For further information about how individual procedures use formatting characters, see the Base SAS Procedures Guide.
specifies in quotation marks a character written to delimit pages. Normally, the delimit character is null, as in this statement:

```sas
options formdlim='';
```

**Details**

When the delimit character is null, a new physical page starts whenever a new page occurs. However, you can conserve paper by allowing multiple pages of output to appear on the same page. For example, this statement writes a line of dashes (- -) where normally a page break would occur:

```sas
options formdlim='-'s;
```

When a new page is to begin, SAS skips a single line, writes a line consisting of the dashes that are repeated across the page, and skips another single line. There is no skip to the top of a new physical page. Resetting FORMDLIM= to null causes physical pages to be written normally again.

*Operating Environment Information:* The syntax that is shown here applies to the OPTIONS statement. On the command line or in a configuration file, the syntax is specific to your operating environment. For details, see the SAS documentation for your operating environment. △

---

**FORMS= System Option**

Specifies the default form that is used for windowing output

**Syntax**

```
FORMS=form-name
```

**Syntax Description**

*form-name*

specifies the name of the form.

**Tip:** To create a customized form, use the FSFORMS command in a windowing environment.

**Details**

The FORMS= system option also customizes output from the PRINT command (when FORM= is omitted) or output from interactive windowing procedures. The default form contains settings that control various aspects of interactive windowing output, including printer selection, text body, and margins.
Operating Environment Information: The syntax that is shown here applies to the OPTIONS statement. On the command line or in a configuration file, the syntax is specific to your operating environment. For details, see the SAS documentation for your operating environment.

GISMAPS= System Option

Specifies the location of the SAS data library that contains SAS/GIS-supplied US Census Tract maps

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Graphics: Driver settings

PROC OPTIONS GROUP= GRAPHICS

See: GISMAPS= System Option in the documentation for your operating environment.

Syntax

GISMAPS=library-specification | path-to-library

Syntax Description

library-specification | path-to-library
specifies either a library or a physical path to a library that contains SAS/GIS-supplied US Census Tract maps.

Operating Environment Information: The syntax shown here applies to the OPTIONS statement. On the command line or in a configuration file, the syntax is specific to your operating environment. For details, see the SAS documentation for your operating environment. A valid library specification and its syntax are specific to your operating environment. On the command line or in a configuration file, the syntax is specific to your operating environment. For details, see the SAS documentation for your operating environment.

See Also

System Option:
“MAPS= System Option” on page 1550

GWINDOW System Option

Controls whether SAS displays SAS/GRAPH output in the GRAPH window of the windowing environment
SAS System Options ∆ HELPENCMD System Option

HELPENCMD System Option

Controls whether SAS uses the English version or the translated version of the keyword list for the command–line Help

Valid in: configuration file, SAS invocation

Category: Environment control: Files

PROC OPTIONS GROUP= HELP

Syntax

HELPENCMD | NOHELPENCMD

Syntax Description

HELPENCMD
specifies that SAS use the English version of the keyword list for the command-line help, although the index will still be displayed with translated keywords. This is the default.

NOHELPENCMD
specifies that SAS use the translated version of the keyword list for the command-line help, if a translated version exists.

Details
Set NOHELPENCMD if you want the command-line help to locate keywords by using the localized terms. By default, all terms on the command line will be read as English.
IBUFSIZE= System Option

Specifies the buffer page size for an index file

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Files: SAS Files

PROC OPTIONS GROUP= SASFILES

Restriction: Specify a page size before the index file is created. After it is created, you cannot change the page size.

Syntax

IBUFSIZE= MIN | MAX | n | nK | nM | nG | nT | hexX

Syntax Description

MIN
sets the page size for an index file to -32767. The IBUFSIZE= option is defined as a signed integer so that negative values can be supplied for internal testing purposes. This might cause unexpected results.

CAUTION:
This setting should be avoided. Use IBUFSIZE=0 to reset IBUFSIZE= to the default value in your operating environment.

MAX
sets the page size for an index file to the maximum possible number. For IBUFSIZE=, the value is 32,767 bytes.

n | nK | nM | nG | nT
specifies the page size to process in multiples of 1 (bytes); 1,024 (kilobytes); 1,048,576 (megabytes); or 1,073,741,824 (gigabytes); or 1,099,511,627,776 (terabytes). For example, a value of 8 specifies 8 bytes, and a value of 3m specifies 3,145,728 bytes.

The default is 0, which causes SAS to use the minimum optimal page size for the operating environment.

hexX
specifies the page size as a hexadecimal value. You must specify the value beginning with a number (0–9), followed by an X. For example, the value 2dx sets the page size to 45 bytes.
Details

An index is an optional SAS file that you can create for a SAS data file in order to provide direct access to specific observations. The index file consists of entries that are organized into hierarchical levels, such as a tree structure, and connected by pointers. When an index is used to process a request, such as for WHERE processing, SAS does a search on the index file in order to rapidly locate the requested record(s).

Typically, you do not need to specify an index page size. However, the following situations could require a different page size:

- The page size affects the number of levels in the index. The more pages there are, the more levels in the index. The more levels, the longer the index search takes. Increasing the page size allows more index values to be stored on each page, thus reducing the number of pages (and the number of levels). The number of pages required for the index varies with the page size, the length of the index value, and the values themselves. The main resource that is saved when reducing levels in the index is I/O. If your application is experiencing a lot of I/O in the index file, increasing the page size might help. However, you must re-create the index file after increasing the page size.

- The index file structure requires a minimum of three index values to be stored on a page. If the length of an index value is very large, you might get an error message that the index could not be created because the page size is too small to hold three index values. Increasing the page size should eliminate the error.

Note: Experimentation is the best way to determine the optimal index page size.

See Also


IMPLMAC System Option

Controls whether SAS allows statement-style macro calls

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Macro: SAS macro

PROC OPTIONS GROUP= MACRO

See: The IMPLMAC System Option in SAS Macro Language: Reference.

INITCMD System Option

Suppresses the Log, Output, and Program Editor windows when you enter a SAS/AF application

Valid in: configuration file, SAS invocation

Category: Environment control: Initialization and operation
PROC OPTIONS GROUP= EXECMODES

Syntax

INITCMD "command-1 <DM-command-n>"

Syntax Description

command-1
specifies any SAS command; the first command in this list must be a command that invokes a window. Some valid values are:

AFA
AF
ASSIST
DESIGN
EIS
FORECAST
HELP
IMAGE
LAB
PHCLINICAL
PHKINETICS
PROJMAN
QUERY
RUNEIS
SQC
XADX.

Restriction: SAS/AF applications may not be available or licensed at all sites.

Interaction: If you specify FORECAST for command-1, you cannot use DM-command-n.

DM-command-n
specifies a valid windowing command or text editor command. Separate multiple commands with semicolons. These commands are processed in sequence. If you use a windowing command that impacts flow, such as the BYE command, it may delay or prohibit processing.

Restriction: Do not use the DM-command-n argument when you enter a SAS/AF application that submits SAS statements or commands during initialization of the application, that is, during autoexec file initialization.

Details

The INITCMD system option suppresses the Log, Output, Program Editor, and Explorer windows when you enter a SAS/AF application, so that the SAS/AF
application is the first screen you see. The suppressed windows do not appear, but you can activate them. You can use the ALTLOG option to direct log output for viewing. If windows are initiated by an autoexec file or the INITSTMT option, the window that is displayed by the INITCMD option is displayed last. When you exit an application that is invoked with the INITCMD option, your SAS session ends.

You can use the INITCMD option in a windowing environment only. Otherwise, the option is ignored and a warning message is issued. If command-1 is not a valid command, the option is ignored and a warning message is issued.

The following SAS execution mode invocation options have precedence over this option:
- DMR
- OBJECTSERVER.

INITCMD has the same precedence as all other SAS execution mode invocation options. If you specify INITCMD with another execution mode invocation option of equal precedence, SAS uses only the last option listed. See “Order of Precedence” on page 1447 for more information on invocation option precedence.

**Examples**

```plaintext
INITCMD "AFA c=mylib.myapp.primary.frame dsname=a.b"
INITCMD "ASSIST; FSVIEW SASUSER.CLASS"
```

---

**INITSTMT= System Option**

Specifies a SAS statement to be executed after any statements in the autoexec file and before any statements from the SYSIN= file

**Valid in:** configuration file, SAS invocation

**Alias:** IS=

**Category:** Environment control: Initialization and operation

**PROC OPTIONS GROUP=** EXECMODES

**See:** INITSTMT= System Option under Windows

---

**Syntax**

INITSTMT='statement'

**Syntax Description**

'statement'

specifies any SAS statement or statements.

**Operating Environment Information:** On the command line or in a configuration file, the syntax is specific to your operating environment. The SYSIN= system option might
not be supported by your operating environment. For details, see the SAS documentation for your operating environment.

**Comparisons**

INITSTMT= specifies the SAS statements to be executed at SAS initialization, and the TERMSTMT= system option specifies the SAS statements to be executed at SAS termination.

**Examples**

Here is an example of using this option on UNIX:

```sas
sas -initstmt '%put you have used the initstmt; data x; x=1;
run;'
```

**See Also**

System Option:

“TERMSTMT= System Option” on page 1613

---

**INVALIDDATA= System Option**

Specifies the value SAS is to assign to a variable when invalid numeric data are encountered

Valid in:  configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category:  Input control: Data Processing

PROC OPTIONS GROUP= INPUTCONTROL

---

**Syntax**

INVALIDDATA='character'

**Syntax Description**

'character'

specifies the value to be assigned, which can be a letter (A through Z, a through z), a period (.), or an underscore (_). The default value is a period.

**Details**

The INVALIDDATA= system option specifies the value that SAS is to assign to a variable when invalid numeric data are read with an INPUT statement or the INPUT function.

*Operating Environment Information:*  The syntax that is shown here applies to the OPTIONS statement. On the command line or in a configuration file, the syntax is
specific to your operating environment. For details, see the SAS documentation for your operating environment.

---

**LABEL System Option**

**Determines whether SAS procedures can use labels with variables**

**Valid in:** configuration file, SAS invocation, OPTIONS statement, SAS System Options window

**Category:** Log and procedure output control: Procedure output

**PROC OPTIONS GROUP=** LISTCONTROL

**Syntax**

LABEL | NOLABEL

**Syntax Description**

**LABEL**

permits SAS procedures to use labels with variables. The LABEL system option must be in effect before the LABEL option of any procedure can be used.

**NOLABEL**

does not allow SAS procedures to use labels with variables. If NOLABEL is specified, the LABEL option of a procedure is ignored.

**Details**

A *label* is a string of up to 256 characters that can be written by certain procedures in place of the variable’s name.

**See Also**

Data Set Option:

“LABEL= Data Set Option” on page 31

---

**_LAST_ = System Option**

**Specifies the most recently created data set**

**Valid in:** configuration file, SAS invocation, OPTIONS statement, SAS System Options window

**Category:** Files: SAS Files

**PROC OPTIONS GROUP=** SASFILES
Syntax

_LAST_ = SAS-data-set

Syntax Description

SAS-data-set
specifies a SAS data set name.

Restriction: No data set options are allowed.

Restriction: Use libref.membername or membername syntax, not a string that is enclosed in quotation marks, to specify a SAS data set name.

Note: You can use quotation marks in the libref.membername or membername syntax if the libref or member name is associated with a SAS/ACCESS engine that supports member names with syntax that requires quoting or name literal (n-literal) specification. For more information, see the SAS/ACCESS documentation. △

Details

By default, SAS automatically keeps track of the most recently created SAS data set. Use the _LAST_= system option to override the default.

_LAST_= is not allowed with data set options.

Operating Environment Information: The syntax that is shown here applies to the OPTIONS statement. On the command line or in a configuration file, the syntax is specific to your operating environment. For details, see the SAS documentation for your operating environment. △

LEFTMARGIN= System Option

Specifies to a printer the size of the margin on the left side of the page

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Log and procedure output control: ODS printing

PROC OPTIONS GROUP= ODSPRINT

Syntax

LEFTMARGIN=margin-size | “margin-size [margin-unit]”

Syntax Description

margin-size
specifies the size of the margin.

**Restriction:** The left margin should be small enough so that the left margin plus the right margin is less than the width of the paper.

**Interactions:** Changing the value of this option may result in changes to the value of the LINESIZE= system option.

**[margin-unit]**

specifies the units for margin-size. The margin-unit can be in for inches or cm for centimeters.

**Default:** inches

**Requirement:** When you specify margin-unit, enclose the entire option value in double quotation marks.

### Details

Note that all margins have a minimum that is dependent on the printer and the paper size.

**Operating Environment Information:** Most SAS system options are initialized with default settings when SAS is invoked. However, the default settings and/or option values for some SAS system options may vary both by operating environment and by site. For details, see the SAS documentation for your operating environment.

For additional information on declaring an ODS printer destination, see ODS statements in *SAS Output Delivery System: User's Guide*.

For additional information on the SAS universal print facility, see “Printing with SAS” in *SAS Language Reference: Concepts*.

### See Also

System Options:

“BOTTOMMARGIN= System Option” on page 1481
“RIGHTMARGIN= System Option” on page 1592
“TOPMARGIN= System Option” on page 1617

---

**LINESIZE= System Option**

**Specifies the line size of SAS procedure output**

**Valid in:** configuration file, SAS invocation, OPTIONS statement, SAS System Options window

**Alias:** LS=

**Category:** Log and procedure output control: SAS log and procedure output

**PROC OPTIONS GROUP=** LOG_LISTCONTROL

**See:** LINESIZE= System Option in the documentation for your operating environment.
**Syntax**

LINESIZE=n | MIN | MAX | hexX

**Syntax Description**

n
specifies the number of lines.

MIN
sets the line size of the SAS procedure output to 64.

MAX
sets the line size of the SAS procedure output to 256.

hexX
specifies the line size of the SAS procedure output as a hexadecimal number. You must specify the value beginning with a number (0–9), followed by an X.
For example, the value 0FAx sets the line size of the SAS procedure output to 250.

**Details**

The LINESIZE= system option specifies the line size (printer line width) for the SAS log and the SAS output that are used by the DATA step and procedures.

*Operating Environment Information:* The syntax that is shown here applies to the OPTIONS statement. On the command line or in a configuration file, the syntax is specific to your operating environment. For details, see the SAS documentation for your operating environment. △

---

**LOGPARM= System Option**

Controls when SAS log files are opened, closed, and, in conjunction with the LOG= system option, how they are named

**Valid in:** configuration file, SAS invocation

**Restriction:** LOGPARM= is valid only in line mode and in batch mode

**Category:** Log and procedure output control: SAS log

**PROC OPTIONS GROUP=** LOGCONTROL

**See:** LOGPARM= System Option in the documentation for your operating environment.

---

**Syntax**

LOGPARM=

“<OPEN= APPEND | REPLACE | REPLACEOLD>
<ROLLOVER= AUTO | NONE | SESSION>
<WRITE= BUFFERED | IMMEDIATE>”
Syntax Description

OPEN=
when a log file already exists, controls how the contents of the existing file are treated.

APPEND
appends the log when opening an existing file. If the file does not already exist, a new file is created.

REPLACE
overwrites the current contents when opening an existing file. If the file does not already exist, a new file is created.

REPLACEOLD
replaces files that are more than one day old. If the file does not already exist, a new file is created.

Operating Environment Information: For z/OS, see the SAS documentation for your operating environment for limitations on the use of OPEN=REPLACEOLD.

Default: REPLACE

ROLOVER=AUTO|NONE|SESSION
controls when or if the SAS log “rolls over,” that is, when the current log is closed and a new one is opened.

AUTO
causes an automatic “rollover” of the log when the directives in the value of the LOG= option change, that is, the current log is closed and a new log file is opened.

Interaction: The name of the new log file is determined by the value of the LOG= system option. If LOG= does not contain a directive, however, the name would never change, so the log would never roll over, even when ROLLOVER=AUTO.

NONE
specifies that rollover does not occur, even when a change occurs in the name that is specified with the LOG= option.

Interaction: If the LOG= value contains any directives, they do not resolve. For example, if Log="#b.log" is specified, the directive “#” does not resolve, and the name of the log file remains "#b.log".

SESSION
at the beginning of each SAS session, opens the log file, resolves directives that are specified in the LOG= system option, and uses its resolved value to name the new log file. During the course of the session, no rollover is performed.

Default: NONE

Interaction: Rollover is triggered by a change in the value of the LOG= option.

Restriction: Rollover will not occur more often than once a minute.

See Also: LOG= system option

WRITE=
specifies when content is written to the SAS log.

BUFFERED
writes content to the SAS log only when a buffer is full in order to increase efficiency.
IMMEDIATE
writes to the SAS log each time that statements are submitted that produce
content for the SAS log.

Default: BUFFERED

Details

The LOGPARM= system option controls the opening and closing of SAS log files when
SAS is operating in batch mode or in line mode. This option also controls the naming of
new log files, in conjunction with the LOG= system option and the use of directives in
the value of LOG=.

Using directives in the value of the LOG= system option allows you to control when
logs are open and closed and how they are named, based on real time events, such as
time, month, day of week, etc.

Operating Environment Information: Under UNIX operating environments, you can
begin directives with either the % symbol or the # symbol. Under Windows and z/OS,
begin directives only with the # symbol. Under OpenVMS, begin directives only with
the % symbol.

The following table contains a list of directives that are valid in LOG= values:

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>%a or #a</td>
<td>Locale's abbreviated day of week</td>
<td>Sun–Sat</td>
</tr>
<tr>
<td>%A or #A</td>
<td>Locale's full day of week</td>
<td>Sunday–Saturday</td>
</tr>
<tr>
<td>%b or #b</td>
<td>Local's abbreviated month</td>
<td>Jan–Dec</td>
</tr>
<tr>
<td>%B or #B</td>
<td>Locale's full month</td>
<td>January–December</td>
</tr>
<tr>
<td>%C or #C</td>
<td>Century number</td>
<td>00–99</td>
</tr>
<tr>
<td>%d or #d</td>
<td>Day of the month</td>
<td>01–31</td>
</tr>
<tr>
<td>%H or #H</td>
<td>Hour</td>
<td>00–23</td>
</tr>
<tr>
<td>%j or #j</td>
<td>Julian day</td>
<td>001–366</td>
</tr>
<tr>
<td>%M or #M</td>
<td>Minutes</td>
<td>00–59</td>
</tr>
<tr>
<td>%m or #m</td>
<td>Month number</td>
<td>01–12</td>
</tr>
<tr>
<td>%n or #n</td>
<td>Current system nodename (without domain name)</td>
<td>00–23</td>
</tr>
<tr>
<td>%s or #s</td>
<td>Seconds</td>
<td>00–59</td>
</tr>
<tr>
<td>%u or #u</td>
<td>Day of week</td>
<td>1= Monday–7=Sunday</td>
</tr>
<tr>
<td>%v or #v</td>
<td>Unique identifier</td>
<td>alphanumeric string that creates a log filename that does not currently exist</td>
</tr>
<tr>
<td>%w or #w</td>
<td>Day of week</td>
<td>0=Sunday–6=Saturday</td>
</tr>
<tr>
<td>%W or #W</td>
<td>Week number (Monday as first day; all days in new year preceding first Monday are in week 00)</td>
<td>00–53</td>
</tr>
<tr>
<td>%y or #y</td>
<td>Year without century</td>
<td>00–99</td>
</tr>
</tbody>
</table>
## SAS System Options

**LOGPARM= System Option**

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>%Y or #Y</td>
<td>Full year</td>
<td>1970–9999</td>
</tr>
<tr>
<td>%%</td>
<td>Percent escape writes a single percent sign in the log filename.</td>
<td>%</td>
</tr>
<tr>
<td>##</td>
<td>Pound escape writes a single pound sign in the log filename.</td>
<td>#</td>
</tr>
</tbody>
</table>

### Operating Environment Information:
See the SAS companion for z/OS for limitations on the length of the log filename under z/OS.

### Examples

**Operating Environment Information:** The LOGPARM= system option is executed when SAS is invoked. When you invoke SAS at your site, the form of the syntax is specific to your operating environment. See the SAS documentation for your operating environment for details.

- **Rolling over the log at a certain time and using directives to name the log according to the time:** If this command is submitted at 9:43 AM, this example creates a log file called test0943.log, and the log rolls over each time the log filename changes. In this example, at 9:44 AM, the test0943.log file will be closed, and the test0944.log file will be opened.

  ```
  sas -log "test%H%M.log" -logparm "rollover=auto"
  ```

- **Preventing log rollover but using directives to name the log:** For a SAS session that begins at 9:34 AM, this example creates a log file named test0934.log, and prevents the log file from rolling over:

  ```
  sas -log "test%H%M.log" -logparm "rollover=session"
  ```

- **Preventing log rollover and preventing the resolution of directives:** This example creates a log file named test%H%M.log, ignores the directives, and prevents the log file from rolling over during the session:

  ```
  sas -log "test%H%M.log" -logparm "rollover=none"
  ```

- **Creating log files with unique identifiers:** This example uses a unique identifier to create a log file with a unique name:

  ```
  sas -log "test%v.log" -logparm "rollover=session"
  ```

  SAS creates a log file called test1.log, if test1.log does not already exist. If test1.log does exist, SAS continues to create filenames in this format—test2.log and so on—until it generates a log filename that does not exist.

  Because %v is not a time-based format, the log file name will never change after it has been generated; therefore, the log will never roll over. In this situation, specifying ROLLOVER=SESSION is equivalent to specifying ROLLOVER=AUTO.
MACRO System Option

 Specifies whether the SAS macro language is available
 Valid in: configuration file, SAS invocation
 Category: Macro: SAS macro
 PROC OPTIONS GROUP= MACRO
 See: The MACRO System Option in SAS Macro Language: Reference.

MAPS= System Option

 Specifies the locations to search for maps
 Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window
 Category: Graphics: Driver settings
 PROC OPTIONS GROUP= GRAPHICS
 See: MAPS= System Option in the documentation for your operating environment

Syntax

MAPS=location-of-maps

Syntax Description

location-of-maps
    specifies either a libref or a physical path.

Operating Environment Information: The syntax shown here applies to the OPTIONS statement. On the command line or in a configuration file, the syntax is specific to your operating environment. For details, see the SAS documentation for your operating environment. Δ

See Also

System Option:
    “GISMAPS= System Option” on page 1536

MAUTOLOCDISPLAY System Option

 Displays the source location of the autocall macros in the log when the autocall macro is invoked
SAS System Options △ MERGENOBY System Option 1551

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window
Category: Macro: SAS macro
PROC OPTIONS GROUP= MACRO
See: The MAUTOLOCDISPLAY System Option in SAS Macro Language: Reference.

MAUTOSOURCE System Option

Determines whether the macro autocall feature is available
Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window
Category: Macro: SAS macro
PROC OPTIONS GROUP= MACRO
See: The MAUTOSOURCE System Option in SAS Macro Language: Reference.

MCOMPILENOTE= System Option

Issues a NOTE to the log upon successful compilation of a macro
Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window
Category: Macro: SAS macro
PROC OPTIONS GROUP= MACRO
See: The MCOMPILENOTE System Option in SAS Macro Language: Reference.

MERGENOBY System Option

Controls whether a message is issued when MERGE processing occurs without an associated BY statement
Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window
Category: Files: SAS Files
PROC OPTIONS GROUP= SASFILES

Syntax
MCCOMPILENOTOD A WERROR | WARN | ERROR
Syntax Description

NOWARN
specifies that no warning message is issued. This is the default.

WARN
specifies that a warning message is issued.

ERROR
specifies that an error message is issued.

MERROR System Option

Controls whether SAS issues a warning message when a macro-like name does not match a macro keyword

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Macro: SAS macro

PROC OPTIONS GROUP= MACRO

See: The MERROR System Option in SAS Macro Language: Reference.

METAAUTORESOURCES= System Option

Identifies what resources to be assigned at SAS startup

Valid in: configuration file, SAS invocation

Category: Communications: Meta Data

PROC OPTIONS GROUP= META

Syntax
METAAUTORESOURCES=resource-identifier

Syntax Description

resource-identifier
is a unique instance identifier that is assigned to a particular LogicalServer, ServerComponent, or ServerContext metadata object in a repository on the SAS Metadata Server. The maximum length is 32,000 characters. Note that if you specify either single or double quotation marks, they are not saved as part of the value.

METAAUTORESOURCES= accepts the following URI formats:

id
is the unique instance identifier that is assigned to the metadata object. An example is `metaautoresources=A3YBDKS4.AH000001`.

typelid
is the metadata object type and the unique instance identifier that is assigned to the metadata object. An example is `metaautoresources=LogicalServer/A3YBDKS4.AH000001`.

type?@search-criteria
is the metadata object type and an attribute value. An example is `metaautoresources=LogicalServer?@Name='My Logical Server'`.

Details
This system option is one of a group of system options that define the default metadata information to use for the SAS session. Usually these values are set at installation time in the configuration file.

`METAAUTORESOURCES=` identifies resources that are to be assigned when you invoke SAS. The resources are defined in a repository on the SAS Metadata Server. For example, in SAS Management Console, you can define a list of librefs (library references) that are associated with the LogicalServer, ServerComponent, or ServerContext object. `METAAUTORESOURCES=` points to the object and assigns the associated libraries at startup.

Note that if the SAS Metadata Server is not available, this option is ignored.

See Also
For information about the SAS Open Metadata Architecture, the SAS Metadata Server, and SAS Metadata Repositories, see `SAS 9.1 Open Metadata Interface: Reference`.

System Options:

```
"METACONNECT= System Option" on page 1553
"METAID= System Option" on page 1557
"METAPASS= System Option" on page 1557
"METAPORT= System Option" on page 1559
"METAPROFILE= System Option" on page 1559
"METAPROTOCOL= System Option" on page 1561
"METAREPOSITORY= System Option" on page 1562
"METASERVER= System Option" on page 1563
"METAUSER= System Option" on page 1564
```

**METACONNECT= System Option**

Identifies the named connection from the metadata user profiles to use as the default values for logging in to the SAS Metadata Server

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Communications: Meta Data
PROC OPTIONS GROUP= META

Syntax

METACONNECT=“named-connection”

Syntax Description

“named-connection”

is a named connection that is contained in the metadata user profiles. An example is
metacconnect=“My Profile”. The maximum length is 256 characters. Quotation
marks are required.

Default: NULL

Details

This system option is one of a group of system options that defines the default metadata
information to use for the SAS session. Usually these values are set at installation time
in the SAS system configuration file. However, the value for METACONNECT= can be
changed at any time.

METACONNECT= identifies which named connection in the metadata user profiles
to use as the default values in order to log in to the metadata server. Each named
connection contains server connection properties such as the name of the host machine
on which the server is invoked, the TCP port, and the user ID and password of the
requesting user.

METACONNECT= searches for the named connection in the metadata user profile
file that is specified by the METAPROFILE= system option. Here is an example:

1 The METAPROFILE= system option specifies the user profile file as
userprofile.xml. The file contains three named connections: A, B, and C. Each
named connection contains properties for logging in to the metadata server.

2 The METACONNECT= system option specifies the named connection as B.

3 The metadata server connection properties that are specified in the named
connection B are loaded from the metadata user profile and used as the values for
logging on to the metadata server.

To create or edit a metadata user profile, use the Metadata Server Connections dialog
box, which you can open by executing the SAS windowing command METACON.

See Also

For information about the SAS Open Metadata Architecture, the SAS Metadata
Server, and SAS Metadata Repositories, see SAS 9.1 Open Metadata Interface
Reference.

System Options:

“METAAUTOREOURCES= System Option” on page 1552
“METAID= System Option” on page 1557
“METAPASS= System Option” on page 1557
“METAPORT= System Option” on page 1559
“METAPROFILE= System Option” on page 1559
METAENCRYPTALG= System Option

Specifies the type of encryption to use when communicating with a SAS Metadata Server

Valid in: configuration file, SAS invocation
Alias: METAENCRYPTALGORITHM=
Category: Communications: Meta Data
PROC OPTIONS GROUP= META

Syntax
METAENCRYPTALG=NONE | RC2 | RC4 | DES | TRIPLEDES | SASPROPRIETARY | SAS

Syntax Description

NONE
specifies that no encryption is used for the SAS Metadata Server connection. This is the default.

RC2
specifies to use the RC2 encryption algorithm that was developed by RSA Security, Inc. The RC2 algorithm uses a variable key-size block cipher to encrypt 64-bit blocks. A single message can be expanded up to 8 bytes. The RC2 key size can range from 8 to 256 bits. RC2 encryption is an alternative to Data Encryption Standard (DES) encryption.

RC4
specifies to use the RC4 encryption algorithm that was developed by RSA Security, Inc. The RC4 algorithm is a variable key-size stream cipher that encrypts 1 byte at a time. The RC4 key size can range from 8 to 2,048 bits.

DES
specifies to use the Data Encryption Standard encryption algorithm that was developed by IBM. The DES algorithm is a block cipher that encrypts data in blocks of 64-bits by using a 56-bit key.

TRIPLEDES
specifies to use the TRIPLEDES encryption algorithm, which processes the DES algorithm sequentially three times on the data. A different 56-bit key is used for each iteration of the encryption. A single message can be expanded up to 8 bytes.

SASPROPRIETARY | SAS
specifies to provide basic encryption services in all operating environments. The SAS algorithm uses a 32-bit key and expands a single message by one-third. No additional product license is required.
Details
In order to specify a proprietary encryption algorithm for the METAENCRYPTALG= system option, you must have a license for SAS/SECURE.

The SAS Integrated Object Model (IOM) supports encryption connections to a SAS Metadata Server. You use the METAENCRYPTALG= option and the METAENCRYPTLEVEL= option to define the type and level of encryption that SAS clients are to use when they communicate with a SAS Metadata Server.

See Also

System Option:
“METAENCRYPTLEVEL= System Option” on page 1556

METAENCRYPTLEVEL= System Option

Specifies what is to be encrypted when communicating with a SAS Metadata Server
Valid in: configuration file, SAS invocation
Category: Communications: Meta Data
PROC OPTIONS GROUP= META

Syntax
METAENCRYPTLEVEL=EVERYTHING | CREDENTIALS

Syntax Description

EVERYTHING
 specifies to encrypt all communication to the SAS Metadata Server.

CREDENTIALS
 specifies to encrypt only login credentials.

Details
In order to specify a level of encryption for the METAENCRYPTLEVEL system option, you must have a license for SAS/SECURE.

The SAS Integrated Object Model (IOM) supports encryption connections to a SAS Metadata Server. You use the METAENCRYPTLEVEL option and the METAENCRYPTALG option to define the level and type of encryption that SAS clients are to use when they communicate with an OMR server.

See Also

System Option:
“METAENCRYPTALG= System Option” on page 1555
**METAID= System Option**

Identifies the current installed version of SAS on the SAS Metadata Server

Valid in: configuration file, SAS invocation  
Category: Communications: Meta Data  
PROC OPTIONS GROUP= META  

Syntax

METAID= id

Syntax Description

id  
is a unique identifier for the current SAS installation. This ID is established automatically during the installation process. An example is `metaid=d3650`. The maximum length is 256 characters.

Details

This system option is one of a group of system options that defines the default metadata information to use for the SAS session. Usually these values are set at installation time in the SAS system configuration file.

METAID= identifies what resources are identified with a particular installation of SAS, that is, what metadata objects are associated with the installation.

See Also

For information on the SAS Open Metadata Architecture, the SAS Metadata Server, and SAS Metadata Repositories, see *SAS 9.1 Open Metadata Interface Reference*.  
System Options:  
"METAAUTORESOURCES= System Option" on page 1552  
"METACONNECT= System Option" on page 1553  
"METAPASS= System Option" on page 1557  
"METAPORT= System Option" on page 1559  
"METAPROFILE= System Option" on page 1561  
"METAPROTOCOL= System Option" on page 1562  
"METAREPOSITORY= System Option" on page 1563  
"METASERVER= System Option" on page 1564  

**METAPASS= System Option**

Sets the default password for the SAS Metadata Server
Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Communications: Meta Data

PROC OPTIONS GROUP= META

Syntax

METAPASS=password

Syntax Description

password

is the password that corresponds to the user identification on the SAS Metadata Server. The maximum length is 512 characters.

Note: To specify an encoded password, use the PWENCODE procedure in order to disguise the text string, then enter the encoded password for METAPASS=. The SAS Metadata Server will decode the encoded password. See the PWENCODE procedure in Base SAS Procedures Guide.

Details

This system option is one of a group of system options that defines the default metadata information to use for the SAS session. Usually these values are set at installation time in the SAS system configuration file.

The network protocol, which is specified with the METAPROTOCOL= system option, determines whether a password is required. If the protocol is COM, a password is not required; if the protocol is BRIDGE, a password is required.

If the password is not specified but is required, and you are running interactively, SAS will display a dialog box in order to acquire the password for the session.

See Also

For information on the SAS Open Metadata Architecture, the SAS Metadata Server, and SAS Metadata Repositories, see SAS 9.1 Open Metadata Interface Reference.

System Options:

“METAAUTHORRESOURCES= System Option” on page 1552
“METACONNECT= System Option” on page 1553
“METAID= System Option” on page 1557
“METAPORT= System Option” on page 1559
“METAPROFILE= System Option” on page 1559
“METAPROTOCOL= System Option” on page 1561
“METAREPOSITORY= System Option” on page 1562
“METASERVER= System Option” on page 1563
“METAUSER= System Option” on page 1564
METAPORT= System Option

Sets the TCP port for the SAS Metadata Server

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Communications: Meta Data

PROC OPTIONS GROUP= META

Syntax

METAPORT=number

Syntax Description

number is the TCP port that the SAS Metadata Server is listening to for connections. An example is metaport=5282.

Default: 0

Range: 0 - 65535

Details

This system option is one of a group of system options that defines the default metadata information to use for the SAS session. Usually these values are set at installation time in the SAS system configuration file. However, the value for METAPORT= can be changed at any time.

See Also

For information on the SAS Open Metadata Architecture, the SAS Metadata Server, and SAS Metadata Repositories, see SAS 9.1 Open Metadata Interface Reference.

System Options:
"METAUTORESOURCES= System Option” on page 1552
"METACONNECT= System Option” on page 1553
"METAID= System Option” on page 1557
"METAPASS= System Option” on page 1557
"METAPROFILE= System Option” on page 1559
"METAPROTOCOL= System Option” on page 1561
"METARepository= System Option” on page 1562
"METASeVER= System Option” on page 1563
"METAUSER= System Option” on page 1564

METAPROFILE= System Option

Identifies the file that contains SAS Metadata Server user profiles
Valid in: configuration file, SAS invocation
Category: Communications: Meta Data
PROC OPTIONS GROUP= META

Syntax
METAPROFILE="XML-document"

Syntax Description

“XML-document”
is the physical location of the XML document that contains metadata user profiles for
logging on to the SAS Metadata Server. The physical name is the name that is
recognized by the operating environment. An example is
metaprofile="!SASROOT\metauser.xml". The maximum length is 32,000
characters. Quotation marks are required.

Details
This system option is one of a group of system options that defines the default metadata
information to use for the SAS session. Usually these values are set at installation time
in the SAS system configuration file.

METAPROFILE= specifies the physical location of the XML document that contains
metadata user profiles. The file defines a list of named connections that contain server
connection properties for logging in to the SAS Metadata Server, such as the name of
the host machine on which the server is invoked, the TCP port, and the user ID and
password of the requesting user.

The METACONNECT= system option then identifies which named connection in the
user profiles to use.

To create or edit a metadata user profile, use the Metadata Server Connections dialog
box, which you can open in one of the following ways:

- selecting File >> New >> Metadata Server Connection from the Explorer
  window.
- executing the SAS windowing command METACON.

See Also
For information about the SAS Open Metadata Architecture, the SAS Metadata
Server, and SAS Metadata Repositories, see SAS 9.1 Open Metadata Interface
Reference.

System Options:
"METAAUTOREOURCES= System Option" on page 1552
"METACONNECT= System Option" on page 1553
"METAID= System Option" on page 1557
"METAPASS= System Option" on page 1557
"METAPORT= System Option" on page 1559
"METAPROTOCOL= System Option" on page 1561
"METAREPOSITORY= System Option" on page 1562
METAPROTOCOL= System Option

Sets the network protocol for communicating with the SAS Metadata Server

Valid in:  configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category:  Communications: Meta Data

PROC OPTIONS GROUP= META

Default:  BRIDGE

Syntax

METAPROTOCOL= BRIDGE | COM

Syntax Description

BRIDGE

specifies that the connection will use the SAS Bridge protocol. This is the default.

COM

specifies that the connection will use Microsoft COM/DCOM services.

Details

This system option is one of a group of system options that defines the default metadata information to use for the SAS session. Usually these values are set at installation time in the SAS system configuration file. However, the value for METAPROTOCOL= can be changed at any time.

The protocol determines which additional connection values are required in order to connect to the metadata server:

<table>
<thead>
<tr>
<th>Option</th>
<th>COM</th>
<th>BRIDGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>METASERVER=</td>
<td>Required</td>
<td>Required</td>
</tr>
<tr>
<td>METAPORT=</td>
<td>Required</td>
<td>Required</td>
</tr>
<tr>
<td>METAUSER=</td>
<td>Not Required</td>
<td>Required *</td>
</tr>
<tr>
<td>METAPASS=</td>
<td>Not Required</td>
<td>Required *</td>
</tr>
</tbody>
</table>

* If not specified and you are running interactively, SAS will display a dialog box in order to acquire these values for the session.
See Also

For information on the SAS Open Metadata Architecture, the SAS Metadata Server, and SAS Metadata Repositories, see *SAS 9.1 Open Metadata Interface Reference.*

System Options:

- “METAAUTORESOURCES= System Option” on page 1552
- “METACONNECT= System Option” on page 1553
- “METAID= System Option” on page 1557
- “METAPASS= System Option” on page 1557
- “METAPORT= System Option” on page 1559
- “METAPROFILE= System Option” on page 1559
- “METAREPOSITORY= System Option” on page 1562
- “METASERVER= System Option” on page 1563
- “METAUSER= System Option” on page 1564

**METAREPOSITORY= System Option**

Sets the default SAS Metadata Repository to use on the SAS Metadata Server

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Communications: Meta Data

PROC OPTIONS GROUP= META

Default: Default

Syntax

**METAREPOSITORY=name**

Syntax Description

**name**

is the name of the SAS Metadata Repository to use. The maximum length is 32,000 characters.

Default: If a repository name is not specified or if the name is invalid, a warning is written to the SAS log and the first repository on the SAS Metadata Server is used.

Details

This system option is one of a group of system options that defines the default metadata information to use for the SAS session. Usually these values are set at installation time in the SAS system configuration file. However, the value for **METAREPOSITORY=** can be changed at any time.

**METAREPOSITORY=** identifies the metadata resources by setting the default SAS Metadata Repository to use.
See Also

For information on the SAS Open Metadata Architecture, the SAS Metadata Server, and SAS Metadata Repositories, see SAS 9.1 Open Metadata Interface Reference.

System Options:

- “METAUTORESOURCES= System Option” on page 1552
- “METACONNECT= System Option” on page 1553
- “METAID= System Option” on page 1557
- “METAPASS= System Option” on page 1557
- “METAPORT= System Option” on page 1559
- “METAPROFILE= System Option” on page 1559
- “METAPROTOCOL= System Option” on page 1561
- “METASERVER= System Option” on page 1563
- “METAUSER= System Option” on page 1564

**METASERVER= System Option**

Sets the address of the SAS Metadata Server

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Communications: Meta Data

PROC OPTIONS GROUP= META

**Syntax**

**METASERVER=** *address*

**Syntax Description**

*address*

is the network IP (Internet Protocol) address of the computer that hosts the SAS Metadata Server. An example is metaserver=d441.na.sas.com. The maximum length is 256 characters.

**Details**

This system option is one of a group of system options that defines the default metadata information to use for the SAS session. Usually these values are set at installation time in the SAS system configuration file. However, the value for METASERVER= can be changed at any time.
METAUSER= System Option

Sets the default user identification for logging onto the SAS Metadata Server

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Communications: Meta Data

PROC OPTIONS GROUP= META

Syntax
METAUSER=id

Syntax Description

id
is the user identification for logging into the SAS Metadata Server. The maximum length is 256 characters.

Details

This system option is one of a group of system options that defines the default metadata information to use for the SAS session. Usually these values are set at installation time in the SAS system configuration file. However, the value for the METAUSER= system option can be changed at any time.

The network protocol, which is specified with the METAPROTOCOL= system option, determines whether a user identification is required. If the protocol is COM, a user identification is not required; if the protocol is BRIDGE, a user identification is required.

If the user identification is not specified and is required, and you are running interactively, SAS will display a dialog box in order to acquire the password for the session.

See Also

For information on the SAS Open Metadata Architecture, the SAS Metadata Server, and SAS Metadata Repositories, see *SAS 9.1 Open Metadata Interface Reference*. System Options:

“METAAUTORESOURCES= System Option” on page 1552
“METACONNECT= System Option” on page 1553
“METAID= System Option” on page 1557
“METAPASS= System Option” on page 1557
“METAPORT= System Option” on page 1559
“METAPROFILE= System Option” on page 1559
“METAPROTOCOL= System Option” on page 1561
“METAREPOSITORY= System Option” on page 1562
“METAUSER= System Option” on page 1564
See Also

For information on the SAS Open Metadata Architecture, the SAS Metadata Server, and SAS Metadata Repositories, see SAS 9.1 Open Metadata Interface Reference.

System Options:

- “METAAUTORESOURCES= System Option” on page 1552
- “METACONNECT= System Option” on page 1553
- “METAID= System Option” on page 1557
- “METAPASS= System Option” on page 1557
- “METAPORT= System Option” on page 1559
- “METAPROFILE= System Option” on page 1559
- “METAPROTOCOL= System Option” on page 1561
- “METAREPOSITORY= System Option” on page 1562
- “METASERVER= System Option” on page 1563

MFILE System Option

Specifies whether MPRINT output is directed to an external file

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Macro: SAS macro

PROC OPTIONS GROUP= MACRO

See: The MFILE System Option in SAS Macro Language: Reference.

MINDELIMITER= System Option

Specifies the character to be used as the delimiter for the macro IN operator

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Macro: SAS macro

PROC OPTIONS GROUP= MACRO

See: The MINDELIMITER System Option in SAS Macro Language: Reference.

MISSING= System Option

Specifies the character to print for missing numeric values
**MLOGIC System Option**

Controls whether SAS traces execution of the macro language processor

**Valid in:** configuration file, SAS invocation, OPTIONS statement, SAS System Options window

**Category:** Macro: SAS macro

**PROC OPTIONS GROUP=** MACRO

**See:** The MLOGIC System Option in *SAS Macro Language: Reference*.

---

**MLOGICNEST System Option**

Displays macro nesting information in the MLOGIC output in the SAS log

**Valid in:** configuration file, SAS invocation, OPTIONS statement, SAS System Options window

**Category:** Macro: SAS macro

**PROC OPTIONS GROUP=** MACRO

**See:** The MLOGICNEST System Option in *SAS Macro Language: Reference*.
**MPRINT System Option**

Displays SAS statements that are generated by macro execution

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Macro: SAS macro

PROC OPTIONS GROUP= MACRO

See: The MPRINT System Option in *SAS Macro Language: Reference*.

---

**MPRINTNEST System Option**

Displays macro nesting information in the MPRINT output in the SAS log

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Macro: SAS macro

PROC OPTIONS GROUP= MACRO

See: The MPRINTNEST System Option in *SAS Macro Language: Reference*.

---

**MRECALL System Option**

Controls whether SAS searches the autocall libraries for a file that was not found during an earlier search

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Macro: SAS macro

PROC OPTIONS GROUP= MACRO

See: The MRECALL System Option in *SAS Macro Language: Reference*.

---

**MSGLEVEL= System Option**

Controls the level of detail in messages that are written to the SAS log

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Log and procedure output control: SAS log

PROC OPTIONS GROUP= LOGCONTROL
Syntax

MSGLEVEL= N | I

Syntax Description

N
prints notes, warnings, and error messages only. This is the default.

I
prints additional notes pertaining to index usage, merge processing, sort utilities, and CEDA usage, along with standard notes, warnings, and error messages.

Details

Some of the conditions under which the MSGLEVEL= system option applies are as follows:

- If MSGLEVEL=I, SAS writes informative messages to the SAS log about index processing. In general, when a WHERE expression is executed for a data set with indexes:
  - if an index is used, a message displays that specifies the name of the index
  - if an index is not used but one exists that could optimize at least one condition in the WHERE expression, messages provide suggestions that describe what you can do to influence SAS to use the index. For example, a message could suggest to sort the data set into index order or to specify more buffers.
  - a message displays the IDXWHERE= or IDXNAME= data set option value if the setting can affect index processing.

- If MSGLEVEL=I, SAS writes a warning to the SAS log when a MERGE statement would cause variables to be overwritten.

- If MSGLEVEL=I, SAS writes a message that indicates which sorting product was used.

- If MSGLEVEL=I, SAS writes a message that indicates when CEDA is being used to process a SAS file.

MSTORED System Option

Determines whether the macro facility searches a specific catalog for a stored, compiled macro

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Macro: SAS macro

PROC OPTIONS GROUP= MACRO

See: The MSTORED System Option in SAS Macro Language: Reference.
MSYMTABMAX= System Option

Specifies the maximum amount of memory that is available to macro variable symbol tables

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Macro: SAS macro

PROC OPTIONS GROUP= MACRO

See: MSYMTABMAX= System Option in the documentation for your operating environment.

See also: The MSYMTABMAX System Option in SAS Macro Language: Reference.

MULTENVAPPL System Option

Controls whether SAS/AF, SAS/FSP, and Base SAS windowing applications use a default on an operating environment specific font selector window

Valid in: configuration file, SAS invocation, OPTIONS statement

Category: Environment control: Initialization and operation

PROC OPTIONS GROUP= EXECMODES

Syntax

MULTENVAPPL | NOMULTENVAPPL

Syntax Description

MULTENVAPPL

specifies that the Select Font window is displayed in the Frame region manager when you set the font for a region title. Use the Select Font window to select different font families, styles, weights, and point sizes.

NOMULTENVAPPL

specifies that SAS/AF and SAS/FSP software display a Font Selector window that is specific to the operating environment in which the application is executing.

Details

The MULTENVAPPL system option enables SAS/AF and SAS/FSP applications, when created, to use the features of the SAS System that are commonly supported in all operating environments. Use this option to ensure portability of applications to multiple operating environments. If NOMULTENVAPPL is in effect when a SAS/AF or SAS/FSP application is created, the application uses features of the SAS System that are more specific to your operating environment. If the application is ported to another environment, a message or a default action may occur, depending on the operating environment.
**MVARSIZE= System Option**

Specifies the maximum size for macro variables that are stored in memory

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Macro: SAS macro

**NEWS= System Option**

Specifies a file that contains messages to be written to the SAS log

Valid in: configuration file, SAS invocation

Category: Environment control: Files

**NOTES System Option**

Writes notes to the SAS log

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Log and procedure output control: SAS log
**PROC OPTIONS GROUP=** LOGCONTROL

**Syntax**

NOTES | NONOTES

**Syntax Description**

**NOTES**

specifies that SAS write notes to the SAS log.

**NONOTES**

specifies that SAS does not write notes to the SAS log. NONOTES does not suppress error and warning messages.

**Details**

You must specify NOTES for SAS programs that you send to SAS for problem determination and resolution.

---

**NUMBER System Option**

Controls the printing of page numbers

**Valid in:** configuration file, SAS invocation, OPTIONS statement, SAS System Options window

**Category:** Log and procedure output control: SAS log and procedure output

**PROC OPTIONS GROUP=** LOG_LISTCONTROL

**Syntax**

NUMBER | NONUMBER

**Syntax Description**

**NUMBER**

specifies that SAS print the page number on the first title line of each page of SAS output.

**NONUMBER**

specifies that SAS not print the page number on the first title line of each page of SAS output.

---

**OBJECTSERVER System Option**

Specifies whether to put the SAS session into DCOM/CORBA server mode
OBS= System Option

Specifies when to stop processing observations or records

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window
Category: Files: SAS Files
PROC OPTIONS GROUP= SASFILES
See: OBS= System Option in the documentation for your operating environment

Syntax

OBS= n | nK | nM | nG | nT | hexX | MIN | MAX

Syntax Description

n | nK | nM | nG | nT
specifies a number to indicate when to stop processing, with n being an integer. Using one of the letter notations results in multiplying the integer by a specific value. That is, specifying K (kilo) multiplies the integer by 1,024; M (mega) multiplies by 1,048,576; G (giga) multiplies by 1,073,741,824; or T (tera) multiplies...
by 1,099,511,627,776. For example, a value of 20 specifies 20 observations or records, while a value of 3m specifies 3,145,728 observations or records.

**hexX**

specifies a number to indicate when to stop processing as a hexadecimal value. You must specify the value beginning with a number (0–9), followed by an X. For example, the hexadecimal value F8 must be specified as 0F8x in order to specify the decimal equivalent of 248. The value 2dx specifies the decimal equivalent of 45.

**MIN**

sets the number to 0 to indicate when to stop processing.

**Interaction:** If OBS=0 and the NOREPLACE option is in effect, then SAS can still take certain actions because it actually executes each DATA and PROC step in the program, using no observations. For example, SAS executes procedures, such as CONTENTS and DATASETS, that process libraries or SAS data sets. External files are also opened and closed. Therefore, even if you specify OBS=0, when your program writes to an external file with a PUT statement, an end-of-file mark is written, and any existing data in the file is deleted.

**MAX**

sets the number to indicate when to stop processing to the maximum number of observations or records, up to the largest eight-byte, signed integer, which is $2^{63}-1$, or approximately 9.2 quintillion. This is the default.

**Details**

OBS= tells SAS when to stop processing observations or records. To determine when to stop processing, SAS uses the value for OBS= in a formula that includes the value for OBS= and the value for FIRSTOBS=. The formula is

\[(\text{obs} - \text{firstobs}) + 1 = \text{results}\]

For example, if OBS=10 and FIRSTOBS=1 (which is the default for FIRSTOBS=), the result is 10 observations or records, that is \((10 - 1) + 1 = 10\). If OBS=10 and FIRSTOBS=2, the result is nine observations or records, that is, \((10 - 2) + 1 = 9\).

OBS= is valid for all steps during your current SAS session or until you change the setting.

You can also use OBS= to control analysis of SAS data sets in PROC steps.

If SAS is processing a raw data file, OBS= specifies the last line of data to read. SAS counts a line of input data as one observation, even if the raw data for several SAS data set observations is on a single line.

**Operating Environment Information:** The syntax that is shown here applies to the OPTIONS statement. On the command line or in a configuration file, the syntax is specific to your operating environment. For details, see the SAS documentation for your operating environment.

**Comparisons**

- An OBS= specification from either a data set option or an INFILE statement option takes precedence over the OBS= system option.
- While the OBS= system option specifies an ending point for processing, the FIRSTOBS= system option specifies a starting point. The two options are often used together to define a range of observations to be processed.
Examples

Example 1: Using OBS= to Specify When to Stop Processing Observations

This example illustrates the result of using OBS= to tell SAS when to stop processing observations. This example creates a SAS data set, executes the OPTIONS statement by specifying FIRSTOBS=2 and OBS=12, then executes the PRINT procedure. The result is 11 observations, that is, \((12 - 2) + 1 = 11\). The result of OBS= in this situation appears to be the observation number that SAS processes last, because the output starts with observation 2, and ends with observation 12, but this is only a coincidence.

```sas
data Ages;
  input Name $ Age;
datalines;
Miguel 53
Brad 27
Willie 69
Marc 50
Sylvia 40
Arun 25
Gary 40
Becky 51
Alma 39
Tom 62
Kris 66
Paul 60
Randy 43
Barbara 52
Virginia 72
;options firstobs=2 obs=12;
proc print data=Ages;
run;
```

Output 8.5 PROC PRINT Output Using OBS= and FIRSTOBS=

<table>
<thead>
<tr>
<th>The SAS System 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obs</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>12</td>
</tr>
</tbody>
</table>

Example 2: Using OBS= with WHERE Processing

This example illustrates the result of using OBS= along with WHERE processing. The example uses the data set that was
created in Example 1, which contains 15 observations, and the example assumes a new SAS session with the defaults FIRSTOBS=1 and OBS=MAX.

First, here is the PRINT procedure with a WHERE statement. The subset of the data results in 12 observations:

```sas
proc print data=Ages;
  where Age LT 65;
run;
```

**Output 8.6 PROC PRINT Output Using a WHERE Statement**

```
The SAS System 1
Obs Name Age
1 Miguel 53
2 Brad 27
4 Marc 50
5 Sylvia 40
6 Arun 25
7 Gary 40
8 Becky 51
9 Alma 39
10 Tom 62
12 Paul 60
13 Randy 43
14 Barbara 52
```

Executing the OPTIONS statement with OBS=10 and the PRINT procedure with the WHERE statement results in 10 observations, that is, \((10 - 1) + 1 = 10\). Note that with WHERE processing, SAS first subsets the data, then applies OBS= to the subset.

```sas
options obs=10;
proc print data=Ages;
  where Age LT 65;
run;
```

**Output 8.7 PROC PRINT Output Using a WHERE Statement and OBS=**

```
The SAS System 2
Obs Name Age
1 Miguel 53
2 Brad 27
4 Marc 50
5 Sylvia 40
6 Arun 25
7 Gary 40
8 Becky 51
9 Alma 39
10 Tom 62
12 Paul 60
```

The result of OBS= appears to be how many observations to process, because the output consists of 10 observations, ending with the observation number 12. However, the result is only a coincidence. If you apply FIRSTOBS=2 and OBS=10 to the subset, the result is nine observations, that is, \((10 - 2) + 1 = 9\). OBS= in this situation is
neither the observation number to end with nor how many observations to process; the value is used in the formula to determine when to stop processing.

options firstobs=2 obs=10;
proc print data=Ages;
   where Age LT 65;
run;

Output 8.8 PROC PRINT Output Using WHERE Statement, OBS=, and FIRSTOBS=

<table>
<thead>
<tr>
<th>Obs</th>
<th>Name</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Brad</td>
<td>27</td>
</tr>
<tr>
<td>4</td>
<td>Marc</td>
<td>50</td>
</tr>
<tr>
<td>5</td>
<td>Sylvia</td>
<td>40</td>
</tr>
<tr>
<td>6</td>
<td>Arun</td>
<td>25</td>
</tr>
<tr>
<td>7</td>
<td>Gary</td>
<td>40</td>
</tr>
<tr>
<td>8</td>
<td>Becky</td>
<td>51</td>
</tr>
<tr>
<td>9</td>
<td>Alma</td>
<td>39</td>
</tr>
<tr>
<td>10</td>
<td>Tom</td>
<td>62</td>
</tr>
<tr>
<td>12</td>
<td>Paul</td>
<td>60</td>
</tr>
</tbody>
</table>

Example 3: Using OBS= When Observations Are Deleted  
This example illustrates the result of using OBS= for a data set that has deleted observations. The example uses the data set that was created in Example 1, with observation 6 deleted. The example also assumes a new SAS session with the defaults FIRSTOBS=1 and OBS=MAX.

First, here is PROC PRINT output of the modified file:

&proc print data=Ages;
run;

Output 8.9 PROC PRINT Output Showing Observation 6 Deleted

<table>
<thead>
<tr>
<th>Obs</th>
<th>Name</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Miguel</td>
<td>53</td>
</tr>
<tr>
<td>2</td>
<td>Brad</td>
<td>27</td>
</tr>
<tr>
<td>3</td>
<td>Willie</td>
<td>69</td>
</tr>
<tr>
<td>4</td>
<td>Marc</td>
<td>50</td>
</tr>
<tr>
<td>5</td>
<td>Sylvia</td>
<td>40</td>
</tr>
<tr>
<td>7</td>
<td>Gary</td>
<td>40</td>
</tr>
<tr>
<td>8</td>
<td>Becky</td>
<td>51</td>
</tr>
<tr>
<td>9</td>
<td>Alma</td>
<td>39</td>
</tr>
<tr>
<td>10</td>
<td>Tom</td>
<td>62</td>
</tr>
<tr>
<td>11</td>
<td>Kris</td>
<td>66</td>
</tr>
<tr>
<td>12</td>
<td>Paul</td>
<td>60</td>
</tr>
<tr>
<td>13</td>
<td>Randy</td>
<td>43</td>
</tr>
<tr>
<td>14</td>
<td>Barbara</td>
<td>52</td>
</tr>
<tr>
<td>15</td>
<td>Virginia</td>
<td>72</td>
</tr>
</tbody>
</table>

Executing the OPTIONS statement with OBS=12, then the PRINT procedure, results in 12 observations, that is, $(12 - 1) + 1 = 12$:

options obs=12;
proc print data=Ages;
The result of OBS= appears to be how many observations to process, because the output consists of 12 observations, ending with the observation number 13. However, if you apply FIRSTOBS=2 and OBS=12, the result is 11 observations, that is \((12 - 2) + 1 = 11\). OBS= in this situation is neither the observation number to end with nor how many observations to process; the value is used in the formula to determine when to stop processing.

```
options firstobs=2 obs=12;
proc print data=Ages;
run;
```

Output 8.11  PROC PRINT Output Using OBS= and FIRSTOBS=

```
<table>
<thead>
<tr>
<th>Obs</th>
<th>Name</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Brad</td>
<td>27</td>
</tr>
<tr>
<td>3</td>
<td>Willie</td>
<td>69</td>
</tr>
<tr>
<td>4</td>
<td>Marc</td>
<td>50</td>
</tr>
<tr>
<td>5</td>
<td>Sylvia</td>
<td>40</td>
</tr>
<tr>
<td>7</td>
<td>Gary</td>
<td>40</td>
</tr>
<tr>
<td>8</td>
<td>Becky</td>
<td>51</td>
</tr>
<tr>
<td>9</td>
<td>Alma</td>
<td>39</td>
</tr>
<tr>
<td>10</td>
<td>Tom</td>
<td>62</td>
</tr>
<tr>
<td>11</td>
<td>Kris</td>
<td>66</td>
</tr>
<tr>
<td>12</td>
<td>Paul</td>
<td>60</td>
</tr>
<tr>
<td>13</td>
<td>Randy</td>
<td>43</td>
</tr>
</tbody>
</table>
```

See Also

Data Set Options:

“FIRSTOBS= Data Set Option” on page 20
“OBS= Data Set Option” on page 32
“REPLACE= Data Set Option” on page 48
System Option:
“FIRSTOBS= System Option” on page 1529

ORIENTATION= System Option

Specifies the paper orientation to use when printing to a printer

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Log and procedure output control: ODS printing

PROC OPTIONS GROUP= ODSPRINT

Syntax

ORIENTATION=PORTRAIT | LANDSCAPE | REVERSEPORTRAIT | REVERSELANDSCAPE

Syntax Description

PORTRAIT
specifies the paper orientation as portrait. This is the default.

LANDSCAPE
specifies the paper orientation as landscape.

REVERSEPORTRAIT
specifies the paper orientation as reverse portrait to enable printing on paper with prepunched holes. The reverse side of the page is printed upside down.

REVERSELANDSCAPE
specifies the paper orientation as reverse landscape to enable printing on paper with prepunched holes. The reverse side of the page is printed upside down.

Details

Changing the value of this option may result in changes to the values of the portable LINESIZE= and PAGESIZE= system options.

Operating Environment Information: Most SAS system options are initialized with default settings when you invoke SAS. However, the default settings for some SAS system options vary both by operating environment and by site. For details, see the SAS documentation for your operating environment.

For additional information on declaring an ODS printer destination, see ODS statements in SAS Output Delivery System: User's Guide.

For additional information on the SAS universal print facility, see “Printing with SAS” in SAS Language Reference: Concepts.
**OVP System Option**

Overprints output lines

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Log and procedure output control: SAS log

PROC OPTIONS GROUP= LOG_LISTCONTROL

**Syntax**

OVP | NOOVP

**Syntax Description**

**OVP**

prints underscores beneath the word in error when SAS encounters an error in a SAS statement. This option may enable overprinting in some procedures.

**NOOVP**

prints dashes on the next line below the word in error when SAS encounters an error in a SAS statement.

**Details**

When output is displayed to a terminal, OVP is overridden and is changed to NOOVP.

---

**PAGEBREAKINITIAL System Option**

Begins the SAS log and listing files on a new page

Valid in: configuration file, SAS invocation

Category: Log and procedure output control: SAS log and procedure output

PROC OPTIONS GROUP= LOG_LISTCONTROL

See: PAGEBREAKINITIAL System Option in the documentation for your operating environment.

**Syntax**

PAGEBREAKINITIAL | NOPAGEBREAKINITIAL

**Syntax Description**

PAGEBREAKINITIAL
begins the SAS log and listing files on a new page.

**NOPAGEBREAKINITIAL**

does not begin the SAS log and listing files on a new page.

**Details**
The PAGEBREAKINITIAL option inserts a page break at the start of the SAS log and listing files. The default behavior is not to begin the SAS log and listing files on a new page.

---

**PAGENO= System Option**

**Resets the page number**

**Valid in:** configuration file, SAS invocation, OPTIONS statement, SAS System Options window

**Category:** Log and procedure output control: Procedure output

**PROC OPTIONS GROUP=** LISTCONTROL

**See:** PAGENO= System Option in the documentation for your operating environment.

---

**Syntax**

PAGENO=\(n \mid nK \mid \text{hex}X \mid \text{MIN} \mid \text{MAX}\)

**Syntax Description**

\(n \mid nK\)

specifies the page number in multiples of 1 (\(n\)); 1,024 (\(nK\)). For example, a value of \(8\) sets the page number to 8 and a value of \(3k\) sets the page number to 3,072.

\(\text{hex}X\)

specifies the page number as a hexadecimal number. You must specify the value beginning with a number (0-9), followed by an X.

For example, the value \(2dx\) sets the page number to 45.

**MIN**

sets the page number to the minimum number, 1.

**MAX**

specifies the maximum page number as the largest signed, four-byte integer that is representable in your operating environment.

**Details**
The PAGENO= system option specifies a beginning page number for the next page of output that SAS produces. Use PAGENO= to reset page numbering during a SAS session.

**Operating Environment Information:** The syntax that is shown here applies to the OPTIONS statement. On the command line or in a configuration file, the syntax is...
PAGESIZE= System Option

Specifies the number of lines that compose a page of SAS output

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Alias: PS=

Category: Log and procedure output control: SAS log and procedure output

PROC OPTIONS GROUP= LOG_LISTCONTROL

See: PAGESIZE= System Option in the documentation for your operating environment.

Syntax

PAGESIZE=n | nK | hexX | MIN | MAX

Syntax Description

n | nK

specifies the number of lines that compose a page in terms of number of pages or units of 1,024 pages.

hex

specifies the number of lines that compose a page as a hexadecimal number. You must specify the value beginning with a number (0–9), followed by an X. For example, the value 2dx sets the number of lines that compose a page to 45 lines.

MIN

sets the number of lines that compose a page to the minimum setting, 15.

MAX

sets the number of lines that compose a page to the maximum setting, 32,767.

Operating Environment Information: The syntax that is shown here applies to the OPTIONS statement. On the command line or in a configuration file, valid values and range vary with your operating environment. For details, see the SAS documentation for your operating environment.

PAPERDEST= System Option

Specifies to a printer the printer bin to receive printed output

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Log and procedure output control: ODS printing
PROC OPTIONS GROUP= ODSPRINT

Syntax

PAPERDEST=printer-bin-name

Syntax Description

printer-bin-name
specifies the bin to receive printed output.

Restriction: Maximum length is 200 characters.

Operating Environment Information: Most SAS system options are initialized with default settings when SAS is invoked. However, the default settings and/or option values for some SAS system options may vary both by operating environment and by site. For details, see the SAS documentation for your operating environment.

For additional information on declaring an ODS printer destination, see ODS statements in SAS Output Delivery System: User's Guide.

For additional information on the SAS universal print facility, see “Printing with SAS” in SAS Language Reference: Concepts.

See Also

System Options:
“PAPERSIZE= System Option” on page 1582
“PAPERSOURCE= System Option” on page 1583
“PAPERTYPE= System Option” on page 1584

PAPERSIZE= System Option

Specifies to a printer the paper size to use

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window
Category: Environment control: Language control

Log and procedure output control: ODS printing

PROC OPTIONS GROUP= LANGUAGECONTROL

ODSPRINT

Syntax

PAPERSIZE=paper_size_name | (“width_value” <,> “height_value”) | (‘width_value’<,> ‘height_value’) | (width_value height_value)
Syntax Description

\textit{paper\_size\_name}  
specifies a predefined paper size. The default is LETTER. Maximum length is 200 characters.

\textbf{Default:} Letter

\textbf{Valid Values:} Refer to the Registry Editor, or use PROC REGISTRY to obtain a listing of supported paper sizes. Additional values can be added.

\textbf{Restriction:} The maximum length is 200 characters.

\textit{\texttt{("width\_value", "height\_value")}}  
specifies paper width and height as positive floating-point values.

\textbf{Default:} inches

\textbf{Range:} \textit{in} or cm for width\_value, height\_value

Details

If you specify a predefined paper size or a custom size that is not supported by your printer, LETTER is used.

Fields specifying values for paper sizes may either be separated by blanks or commas.

\textbf{Note:} Changing the value of this option may result in changes to the values of the portable LINESIZE= and PAGESIZE= system options. △

\textit{Operating Environment Information:} Most SAS system options are initialized with default settings when SAS is invoked. However, the default settings and/or option values for some SAS system options may vary both by operating environment and by site. For details, see the SAS documentation for your operating environment. △

For additional information on declaring an ODS printer destination, see ODS statements in \textit{SAS Output Delivery System: User's Guide}

For additional information on the SAS universal print facility, see “Printing with SAS” in \textit{SAS Language Reference: Concepts}.

See Also

System Options:

“PAPERDEST= System Option” on page 1581
“PAPERSOURCE= System Option” on page 1583
“PAPERTYPE= System Option” on page 1584

\textbf{PAPERSOURCE=} System Option

\textit{Specifies to a printer the paper bin to use for printing}

\textbf{Valid in:} configuration file, SAS invocation, OPTIONS statement, SAS System Options window

\textbf{Category:} Log and procedure output control: ODS printing

\textbf{PROC OPTIONS GROUP=} ODSPRINT
PAPERTYPE= System Option

Specifies to a printer the type of paper to use for printing

Valid in: configuration file, SAS invocation, OPTIONS statement SAS System Options window

Category: Log and procedure output control: ODS printing

PROC OPTIONS GROUP= ODSPRINT

Syntax

PAPERTYPE= paper-type-string

Syntax Description

paper-type-string
specifies the type of paper. Maximum length is 200.

Range: Values vary by site and operating environment.
**Default:** Values vary by site and operating environment.

**Operating Environment Information:** For instructions on how to specify the type of paper, see the SAS documentation for your operating environment. There is a very large number of possible values for this option.

For additional information on declaring an ODS printer destination, see ODS statements in *SAS Output Delivery System: User’s Guide*.

For additional information on the SAS universal print facility, see “Printing with SAS” in *SAS Language Reference: Concepts*.

**See Also**

System Options:

- “PAPERDEST= System Option” on page 1581
- “PAPERSIZE= System Option” on page 1582
- “PAPERSOURCE= System Option” on page 1583

---

**PARM= System Option**

**Specifies a parameter string that is passed to an external program**

**Valid in:** configuration file, SAS invocation, OPTIONS statement, SAS System Options window

**Category:** Environment control: Files

**PROC OPTIONS GROUP=** ENVFILES

---

**Syntax**

PARM=’<string>‘

**Syntax Description**

’<string>‘ specifies a character string that contains a parameter.

**Examples**

This statement passes the parameter X=2 to an external program:

```sql
options parm='x=2';
```

**Operating Environment Information:** Other methods of passing parameters to external programs depend on your operating environment and on whether you are running in interactive line mode or batch mode. For details, see the SAS documentation for your operating environment.
PARMCARDS= System Option

Specifies the file reference to use as the PARMCARDS file

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Environment control: Files

PROC OPTIONS GROUP= ENVFILES

See: PARMCARDS= System Option in the documentation for your operating environment.

Syntax

PARMCARDS=file-ref

Syntax Description

file-ref specifies the file reference to use as the PARMCARDS= file.

Details

The PARMCARDS= system option specifies the file reference of a file that SAS opens when it encounters a PARMCARDS (or PARMCARDS4) statement in a procedure.

SAS writes all data lines after the PARMCARDS (or PARMCARDS4) statement to the file until it encounters a delimiter line of either one or four semicolons. The file is then closed and made available to the procedure to read. There is no parsing or macro expansion of the data lines.

Operating Environment Information: The syntax shown here applies to the OPTIONS statement. On the command line or in a configuration file, the syntax is specific to your operating environment. For details, see the SAS documentation for your operating environment. ∆

PRINTERPATH= System Option

The PRINTERPATH= option controls which Universal Printing printer will be used for printing.

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Log and procedure output control: ODS printing

PROC OPTIONS GROUP= ODSPRINT

Syntax

PRINTERPATH=('printer-name' <fileref>)
Syntax Description

'printer-name'

must be one of the printers defined in the Registry Editor under Core ▶ Printing ▶ Printers

Requirement: When the printer name contains blanks, you must enclose it in quotation marks.

fileref

is an optional fileref. If a fileref is specified, it must be defined with a FILENAME statement or external allocation. If a fileref is not specified, the default output destination defined in the Registry Editor under Core ▶ Printing ▶ Printers ▶ Printer Setup ▶ Output is used. Parentheses are required only when a fileref is specified.

Details

If the PRINTERPATH= option is not a null string, then Universal Printing will be used. If the PRINTERPATH= option does not specify a valid Universal Printing printer, then the default Universal Printer will be used.

Comparisons

A related system option SYSPRINT specifies which operating system printer will be used for printing. PRINTERPATH= specifies which Universal Printing printer will be used for printing.

The operating system printer specified by the SYSPRINT option is used when the PRINTERPATH= option is a null string.

Examples

The following example specifies an output destination that is different from the default:

   options PRINTERPATH=(corelab out);
   filename out 'your_file';

Operating Environment Information: In some operating environments, such as the PC, the SYSPRINT option enables operating-environment printing and disables SAS printing by setting the PRINTERPATH= option to a null string. Also, in some operating environments, setting the PRINTERPATH= option might not change the setting of the PMENU print button, which might continue to use operating environment printing. See the SAS documentation for your operating environment for more information.

For additional information on declaring an ODS printer destination, see ODS statements in SAS Output Delivery System: User’s Guide. 

For additional information on the SAS universal print facility, see “Printing with SAS” in SAS Language Reference: Concepts.

PRINTINIT System Option

Initializes the SAS file
Valid in: configuration file, SAS invocation
Category: Log and procedure output control: Procedure output
PROC OPTIONS GROUP= LISTCONTROL
See: PRINTINIT System Option in the documentation for your operating environment.

Syntax
PRINTINIT | NOPRINTINIT

Syntax Description

PRINTINIT
empties the SAS output file and resets the file attributes upon initialization.
Tip: Specifying PRINTINIT causes the SAS output file to be emptied even when output is not generated.

NOPRINTINIT
preserves the existing output file if no new output is generated. This is the default.
Tip: Specifying NOPRINTINIT causes the SAS output file to be overwritten only when new output is generated.

Details

Operating Environment Information: The behavior of the PRINTINIT system option depends on your operating environment. For additional information, see the SAS documentation for your operating environment.

PRINTMSGLIST System Option

Controls the printing of extended lists of messages to the SAS log
Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window
Category: Log and procedure output control: SAS log
PROC OPTIONS GROUP= LOGCONTROL

Syntax
PRINTMSGLIST | NOPRINTMSGLIST

Syntax Description

PRINTMSGLIST
prints the entire list of messages to the SAS log.
NOPRINTMSGLIST
prints only the top-level message to the SAS log.

**Details**

For Version 7 and later versions, the return code subsystem allows for lists of return codes. All of the messages in a list are related, in general, to a single error condition, but give different levels of information. This option enables you to see the entire list of messages or just the top-level message. The default is set to "ON."

---

**QUOTELENMAX System Option**

Specifies that SAS write to the SAS log a warning about the maximum length for strings in quotation marks

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Environment control: Error handling

**PROC OPTIONS GROUP=** ERRORHANDLING

**Syntax**

QUOTELENMAX | NOQUOTELENMAX

**Syntax Description**

QUOTELENMAX specifies that SAS write to the SAS log a warning for the maximum length for strings in quotation marks to the SAS log

NOQUOTELENMAX specifies that SAS does not write a warning for the maximum length for strings in quotation marks to the SAS log

**Details**

If a string in quotation marks is too long, SAS writes the following warning to the SAS log:

WARNING 32-169: The quoted string currently being processed has become more than 262 characters long. You may have unbalanced quotation marks.

If you are running a program that has long strings in quotation marks, and you do not want to see this warning, use the NOQUOTELENMAX system option to turn off the warning.
REPLACE System Option

Controls whether you can replace permanently stored SAS data sets

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Files: SAS Files

PROC OPTIONS GROUP= SASFILES

Syntax

REPLACE | NOREPLACE
Syntax Description

REPLACE
   specifies that a permanently stored SAS data set can be replaced with another SAS data set of the same name.

NOREPLACE
   specifies that a permanently stored SAS data set cannot be replaced with another SAS data set of the same name. This prevents the accidental replacement of existing SAS data sets.

Details
This option has no effect on data sets in the WORK library, even if you use the WORKTERM= system option to store the WORK library files permanently.

Comparisons
The REPLACE= data set option overrides the REPLACE system option.

See Also

System Option:
   “WORKTERM System Option” on page 1630

Data Set Option:
   “REPLACE= Data Set Option” on page 48

REUSE= System Option

Specifies whether or not SAS reuses space when observations are added to a compressed SAS data set

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Files: SAS Files

PROC OPTIONS GROUP= SASFILES

Syntax
REUSE=YES | NO

Syntax Description

YES
tracks free space and reuses it whenever observations are added to an existing compressed data set.

**NO**

does not track free space. This is the default.

**Details**

If space is reused, observations that are added to the SAS data set are inserted wherever enough free space exists, instead of at the end of the SAS data set.

Specifying REUSE=NO results in less efficient usage of space if you delete or update many observations in a SAS data set. However, the APPEND procedure, the FSEDIT procedure, and other procedures that add observations to the SAS data set continue to add observations to the end of the data set, as they do for uncompressed SAS data sets.

You cannot change the REUSE= attribute of a compressed SAS data set after it is created. This means that space is tracked and reused in the compressed SAS data set according to the REUSE= value that was specified when the SAS data set was created, not when you add and delete observations. Even with REUSE=YES, the APPEND procedure will add observations at the end.

**Operating Environment Information:** The syntax that is shown here applies to the OPTIONS statement. On the command line or in a configuration file, the syntax is specific to your operating environment. For details, see the SAS documentation for your operating environment.

**Comparisons**

The REUSE= data set option overrides the REUSE= system option.

**PERFORMANCE NOTE:** When using COMPRESS=YES and REUSE=YES system options settings, observations cannot be addressed by observation number.

Note that REUSE=YES takes precedence over the POINTOBS=YES data set option setting.

**See Also**

System Option:

“COMPRESS= System Option” on page 1500

Data Set Options:

“COMPRESS= Data Set Option” on page 14

“REUSE= Data Set Option” on page 49

---

**RIGHTMARGIN= System Option**

Specifies the size of the margin at the right side of the page for printed output directed to the ODS printer destination

**Valid in:** configuration file, SAS invocation, OPTIONS statement, SAS System Options window

**Category:** Log and procedure output control: ODS printing

**PROC OPTIONS GROUP=** ODSPRINT
**Syntax**

RIGHTMARGIN=margin-size | “margin-size [margin-unit]”

**Syntax Description**

*margin-size*  
specifies the size of the margin.  
**Restriction:**  The right margin should be small enough so that the left margin plus the right margin is less than the width of the paper.  
**Interactions:**  Changing the value of this option may result in changes to the value of the LINESIZE= system option.

*margin-unit*  
specifies the units for margin-size. The margin-unit can be *in* for inches or *cm* for centimeters.  
**Default:**  inches  
**Requirement:**  When you specify margin-unit, enclose the entire option value in double quotation marks.

**Details**

Note that all margins have a minimum that is dependent on the printer and the paper size.

*Operating Environment Information:*  Most SAS system options are initialized with default settings when SAS is invoked. However, the default settings and/or option values for some SAS system options may vary both by operating environment and by site. For details, see the SAS documentation for your operating environment. △

For additional information on declaring an ODS printer destination, see the ODS statements in *SAS Output Delivery System: User’s Guide*

**See Also**

System Options:  
“BOTTOMMARGIN= System Option” on page 1481  
“LEFTMARGIN= System Option” on page 1544  
“TOPMARGIN= System Option” on page 1617

**RSASUSER System Option**

Controls access to the SASUSER library  
**Valid in:**  configuration file, SAS invocation
Category: Environment control: Files

PROC OPTIONS GROUP= ENVFILES

See: RSASUSER System Option in the documentation for your operating environment.

---

**Syntax**

RSASUSER | NORSASUSER

**Syntax Description**

**RSASUSER**
- opens the SASUSER library in input (that is, read-only) mode.

**NORSASUSER**
- opens the SASUSER library in update (that is, read-write) mode.

**Details**

The RSASUSER system option is useful for sites that use a single SASUSER library for all users and want to prevent users from modifying it. However, it is not useful when users use SAS/ASSIST software, because SAS/ASSIST requires writing to the SASUSER library. For details, see the SAS documentation for your operating environment.

---

**S= System Option**

**Syntax**

S=n | nK | nM | nG | nT | hexX | MIN | MAX

**Syntax Description**

\( n \) | \( nK \) | \( nM \) | \( nG \) | \( nT \) specifies the length of statements and data in terms of 1 (bytes); 1,024 (kilobytes); 1,048,576 (megabytes); 1,073,741,824 (gigabytes); or 1,099,511,627,776 (terabytes).

For example, a value of 8 specifies 8 bytes, and a value of 3m specifies 3,145,728 bytes.

**hexX**
specifies the length of statements and data as a hexadecimal number. You must specify the value beginning with a number (0–9), followed by an X. For example, the value 2dx sets the length of statements and data to 45.

**MIN**
sets the length of statements and data to 0, and requires SAS to use a default value.

**MAX**
sets the length of statements and data to the largest unsigned, four-byte integer that is representable in your operating environment.

**Details**
Input can be from either fixed- or variable-length records. Both fixed-length and variable-length records can be either unsequenced or sequenced. Unsequenced records do not contain sequence fields. Fixed-length sequenced records contain sequence fields at the end of each record. Variable-length sequenced records contain sequence fields at the beginning of each record.

SAS determines whether the input contains sequence numbers that are based on the value of S. If S=0 and you have fixed-length records, SAS inspects the last \( n \) columns (where \( n \) is the value of the SEQ= option) of the first sequence field, which is at the end of the first line of input. If those columns contain numeric characters, SAS assumes that the file contains sequence fields and ignores the last eight columns of each line.

If \( S \geq 0 \) or MAX and you have fixed-length records, SAS uses that value as the length of the source or data to be scanned, ignores everything beyond that length on each line, and does not look for sequence numbers.

If S=0 and you have variable-length records, SAS inspects the last \( n \) columns (where \( n \) is the value of SEQ=) of the first sequence field, which is at the beginning of the first line of input. If those columns contain numeric characters, SAS assumes the file contains sequence fields and ignores the first eight columns of each line.

If \( S \geq 0 \) or MAX and you have variable-length records, SAS uses that value as the starting column of the source or data to be scanned, ignores everything before that length on each line, and does not look for sequence numbers.

**Comparisons**
The S= system option operates exactly like the S2= system option except that S2= controls input only from a %INCLUDE statement, an autoexec file, or an autocall macro file.

*Operating Environment Information:* The syntax that is shown here applies to the OPTIONS statement. On the command line or in a configuration file, the syntax is specific to your operating environment. For details, see the SAS documentation for your operating environment.

**See Also**

System Options:

“S2= System Option” on page 1596

“SEQ= System Option” on page 1598
**S2= System Option**

Specifies the length of secondary source statements

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Input control: Data Processing

PROC OPTIONS GROUP= INPUTCONTROL

See: S2= System Option in the documentation for your operating environment.

**Syntax**

S2=S | n

**Syntax Description**

S  
uses the current value of the S= system option to compute the record length of text that comes from a %INCLUDE statement, an autoexec file, or an autocall macro file.

n  
uses the value of n to compute the record length of text that comes from a %INCLUDE statement, an autoexec file, or an autocall macro file.

**Comparisons**

The S2= system option operates exactly like the S= system option except that the S2= option controls input from only a %INCLUDE statement, an autoexec file, or an autocall macro file.

*Operating Environment Information:* The syntax that is shown here applies to the OPTIONS statement. On the command line or in a configuration file, the syntax is specific to your operating environment. For details, see the SAS documentation for your operating environment.

**See Also**

System Options:

“S= System Option” on page 1594

“SEQ= System Option” on page 1598

**SASAUTOS= System Option**

Specifies the autocall macro library
**SASHELP= System Option**

Specifies the location of the SASHELP library

Valid in: configuration file, SAS invocation

Category: Environment control: Files

PROC OPTIONS GROUP= ENVFILES

See: SASHELP= System Option in the documentation for your operating environment.

---

**Syntax**

SASHELP=library-specification

**Syntax Description**

*library-specification* identifies an external library.

**Details**

The SASHELP= system option is set during the installation process and normally is not changed after installation.

*Operating Environment Information:* A valid external library specification is specific to your operating environment. On the command line or in a configuration file, the syntax is specific to your operating environment. For details, see the SAS documentation for your operating environment.

---

**SASMSTORE= System Option**

Specifies the libref of a SAS data library that contains a catalog of stored, compiled SAS macros

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Environment control: Files

Macro: SAS macro

PROC OPTIONS GROUP= ENVFILES MACRO

See: SASAUTOS= System Option in the documentation for your operating environment.

See also: The SASAUTOS System Option in *SAS Macro Language: Reference*. 
SASUSER= System Option

Specifies the name of the SASUSER library
Valid in: configuration file, SAS invocation
Category: Environment control: Files
PROC OPTIONS GROUP= ENVFILES
See: SASUSER= System Option in the documentation for your operating environment.

Syntax
SASUSER=SAS-data-library

Syntax Description

SAS-data-library
  identifies a SAS data library that contains a user’s profile catalog.

Details
The library and catalog are created automatically by SAS; you do not have to create them explicitly.

Operating Environment Information: A valid library specification and its syntax are specific to your operating environment. On the command line or in a configuration file, the syntax is specific to your operating environment. For details, see the SAS documentation for your operating environment.

SEQ= System Option

Specifies the length of the numeric portion of the sequence field in input source lines or data lines
Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window
Category: Input control: Data Processing
PROC OPTIONS GROUP= INPUTCONTROL

Syntax
SEQ=n | MIN | MAX | hexX
Syntax Description

\[ n \]

specifies the length in terms of bytes.

\[ \text{MIN} \]

sets the minimum length to 1.

\[ \text{MAX} \]

sets the maximum length to 8.

\[ \text{Tip: When SEQ=8, all eight characters in the sequence field are assumed to be numeric.} \]

\[ \text{hexX} \]

specifies the length as a hexadecimal. You must specify the value beginning with a number (0–9), followed by an X.

Details

Unless the S= or S2= system option specifies otherwise, SAS assumes an eight-character sequence field; however, some editors place some alphabetic information (for example, the file name) in the first several characters. The SEQ= value specifies the number of digits that are right justified in the eight-character field. For example, if you specify SEQ=5 for the sequence field AAA00010, SAS looks at only the last five characters of the eight-character sequence field and, if the characters are numeric, treats the entire eight-character field as a sequence field.

See Also

System Options:

“S= System Option” on page 1594
“S2= System Option” on page 1596

SERROR System Option

Controls whether SAS issues a warning message when a defined macro variable reference does not match a macro variable

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Macro: SAS macro

PROC OPTIONS GROUP= MACRO

See: The SERROR System Option in *SAS Macro Language: Reference*. 

SETINIT System Option

Controls whether site license information can be altered
**Valid in:** configuration file, SAS invocation  
**Category:** System administration: Installation  
**PROC OPTIONS GROUP=** INSTALL

---

**Syntax**

SETINIT | NOSETINIT

**Syntax Description**

**SETINIT**

in a non-windowing environment, enables you to change license information by running the SETINIT procedure.

**NOSETINIT**

does not allow you to alter site license information after installation.

---

**Details**

SETINIT is set in the installation process and is not normally changed after installation. The SETINIT option is valid only in a non-windowing SAS session.

---

**SKIP= System Option**

**Specifies the number of lines to skip at the top of each page of SAS output**

**Valid in:** configuration file, SAS invocation, OPTIONS statement, SAS System Options window  
**Category:** Log and procedure output control: Procedure output  
**PROC OPTIONS GROUP=** LISTCONTROL

---

**Syntax**

SKIP=n | hexX | MIN | MAX

**Syntax Description**


\[n\]

specifies the range of lines to skip from 0 to 20.

**MIN**

sets the number of lines to skip to 0, so no lines are skipped.

**MAX**

sets the number of lines to skip to 20.
\textit{hex}

specifies the number of lines to skip as a hexadecimal number. You must specify the value beginning with a number (0–9), followed by an X.

For example, the value \texttt{0ax} specifies to skip 10 lines.

**Details**

The location of the first line is relative to the position established by carriage control or by the forms control buffer on the printer. Most sites define this so that the first line of a new page begins three or four lines down the form. If this spacing is sufficient, specify \texttt{SKIP=0} so that additional lines are not skipped.

The \texttt{SKIP=} value does not affect the maximum number of lines printed on each page, which is controlled by the \texttt{PAGESIZE=} system option.

---

**SOLUTIONS System Option**

Specifies whether the \texttt{SOLUTIONS} menu choice appears in all SAS windows and whether the \texttt{SOLUTIONS} folder appears in the SAS Explorer window tree view.

*Valid in:* configuration file, SAS invocation

*Category:* Environment control: Display

*PROC OPTIONS GROUP=* ENVDISPLAY

**Syntax**

\texttt{SOLUTIONS | NOSOLUTIONS}

**Syntax Description**

**SOLUTIONS**

specifies that the \texttt{SOLUTIONS} menu choice appears in all SAS windows and that the \texttt{SOLUTIONS} folder appears in the SAS Explorer window tree view.

**NOSOLUTIONS**

specifies that the \texttt{SOLUTIONS} menu choice does not appear in all SAS windows and the \texttt{SOLUTIONS} folder does not appear in the SAS Explorer window tree view.

---

**SORTDUP= System Option**

Controls the \texttt{SORT} procedure’s application of the \texttt{NODUP} option to physical or logical records.

*Valid in:* configuration file, SAS invocation, OPTIONS statement, SAS System Options window

*Category:* Sort: Procedure options

*PROC OPTIONS GROUP=* SORT
Syntax
SORTDUP=PHYSICAL | LOGICAL

Syntax Description

PHYSICAL
removes duplicates based on all the variables that are present in the data set. This is the default.

LOGICAL
removes duplicates based on only the variables remaining after the DROP= and KEEP= data set options are processed.

Interactions: The SORTDUP option is relevant only when DROP= or KEEP= data set options are specified on the input data set for the SORT procedure. The NODUP option is a SORT procedure option that removes duplicate records. For further information about how NODUP and SORTDUP interact, see the Base SAS Procedures Guide.

See Also
System Options:
“SORTSEQ= System Option” on page 1603
“SORTSIZE= System Option” on page 1603
“The SORT Procedure” in Base SAS Procedures Guide

SORTEQUALS System Option

Controls how PROC SORT orders observations with identical BY values in the output data set

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System OPTIONS window
Category: Sort: Procedure options

PROC OPTIONS GROUP= SORT

Syntax
SORTEQUALS | NOSORTEQUALS

SORTEQUALS
specifies that observations with identical BY variable values are to retain the same relative positions in the output data set as in the input data set.

NOSORTEQUALS
specifies that no resources be used to control the order of observations with identical BY variable values in the output data set.

**Interaction:** To achieve the best sorting performance when using the THREADS= system option, specify THREADS=YES and NOSORTEQUALS.

**Tip:** To save resources, use NOSORTEQUALS when you do not need to maintain a specific order of observations with identical BY variable values.

**Comparisons**

The SORTEQUALS and NOSORTEQUALS system options set the sorting behavior of PROC SORT for your SAS session. The EQUAL or NOEQUAL option in the PROC SORT statement overrides the setting of the system option for an individual PROC step and specifies the sorting behavior for that PROC step only.

**See Also**

Statement Options:
- EQUALS option for the PROC SORT statement in *Base SAS Procedures Guide*.

System Options:
- “THREADS System Option” on page 1615

---

**SORTSEQ= System Option**

Specifies a language-specific collation sequence for the SORT procedure to use in the current SAS session.

**Valid in:** configuration file, SAS invocation, OPTIONS statement, SAS System Options window

**Category:** Sort: Procedure options

**PROC OPTIONS GROUP=** SORT

**See:** The SORTSEQ= system option in *SAS National Language Support (NLS): User’s Guide*

---

**SORTSIZE= System Option**

Specifies the amount of memory that is available to the SORT procedure.

**Valid in:** configuration file, SAS invocation, OPTIONS statement, SAS System Options window

**Category:** Sort: Procedure options
- System administration: Memory

**PROC OPTIONS GROUP=** MEMORY
- SORT

**See:** SORTSIZE= System Option in the documentation for your operating environment.
Syntax

SORTSIZE= n | nK | nM | nG | nT | hex | MIN | MAX

Syntax Description

\( n \ | \ nK \ | \ nM \ | \ nG \ | \ nT \)

specifies the amount of memory in terms of 1 (byte); 1,024 (kilobytes); 1,048,576 (megabytes); 1,073,741,824 (gigabytes); or 1,099,511,627,776 (terabytes). For example, a value of 4000 specifies 4,000 bytes and a value of 2m specifies 2,097,152 bytes. If \( n=0 \), the sort utility uses its default. Valid values for SORTSIZE range from 0 to 9,223,372,036,854,775,807.

\( \text{hex} \)

specifies the amount of memory as a hexadecimal number. This number must begin with a number (0-9), followed by an X. For example, \text{0fffx} specifies 4095 bytes of memory.

MIN

specifies the minimum amount of memory available.

MAX

specifies the maximum amount of memory available.

Operating Environment Information: Values for MIN and MAX will vary, depending on your operating environment. For details, see the SAS documentation for your operating environment.

Details

Generally, the value of the SORTSIZE= system option should be less than the physical memory available to your process. If the SORT procedure needs more memory than you specify, the system creates a temporary utility file.

PERFORMANCE NOTE: Proper specification of SORTSIZE= can improve sort performance by restricting the swapping of memory that is controlled by the operating environment.

See Also

System Option:

“SUMSIZE= System Option” on page 1607

“The SORT procedure” in the SAS documentation for your operating environment

SOURCE System Option

Controls whether SAS writes source statements to the SAS log
SAS System Options

**SOURCE2 System Option**

Controls whether SAS writes secondary source statements from included files to the SAS log

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Log and procedure output control: SAS log

PROC OPTIONS GROUP= LOGCONTROL

---

**Syntax**

SOURCE2 | NOSOURCE2

**Syntax Description**

**SOURCE2**

writes to the SAS log secondary source statements from files that have been included by %INCLUDE statements.

**NOSOURCE2**

does not write secondary source statements to the SAS log.

---

**SOURCE System Option**

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Log and procedure output control: SAS log

PROC OPTIONS GROUP= LOGCONTROL

---

**Syntax**

SOURCE | NOSOURCE

**Syntax Description**

**SOURCE**

writes SAS source statements to the SAS log.

**NOSOURCE**

does not write SAS source statements to the SAS log.

**Details**

The SOURCE system option does not affect whether statements from the autoexec file, from a file read with %INCLUDE, or from an autocall macro are printed in the SAS log.

*Note:* SOURCE must be in effect when you execute SAS programs that you want to send to SAS for problem determination and resolution.
Details

Note: SOURCE2 must be in effect when you execute SAS programs that you want to send to SAS for problem determination and resolution.

SPOOL System Option

Controls whether SAS writes SAS statements to a utility data set in the WORK data library

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Input control: Data Processing

PROC OPTIONS GROUP= INPUTCONTROL

Syntax

SPOOL | NOSPOOL

Syntax Description

SPOOL
writes SAS statements to a utility data set in the WORK data library for later use by a %INCLUDE or %LIST statement, or by the RECALL command, within a windowing environment.

NOSPOOL
does not write SAS statements to a utility data set. Specifying NOSPOOL accelerates execution time, but you cannot use the %INCLUDE and %LIST statements to resubmit SAS statements that were executed earlier in the session.

Examples

Specifying SPOOL is especially helpful in interactive line mode because you can resubmit a line or lines of code by referring to the line numbers. Here is an example of code including line numbers:

```
00001 data test;
00002 input w x y z;
00003 datalines;
00004 411.365 101.945 323.782 512.398
00005 ;
```

If SPOOL is in effect, you can resubmit line number 1 by submitting this statement:

```
%inc 1;
```

You also can resubmit a range of lines by placing a colon (:) or dash (-) between the line numbers. For example, these statements resubmit lines 1 through 3 and 4 through 5 of the above example:
%inc 1:3;
%inc 4-5;

STARTLIB System Option

Allows previous library references (librefs) to persist in a new SAS session

Valid in: configuration file, SAS invocation
Category: Files: External files
PROC OPTIONS GROUP= EXTFILES

Syntax

STARTLIB | NOSTARTLIB

Syntax Description

STARTLIB
causes library references (librefs) specified in a previous SAS session to persist in a new SAS session. This option only affects librefs that were assigned in the New Library window and selected in that window to be enabled at startup.

NOSTARTLIB
does not autoassign librefs at startup. This option only affects librefs that were assigned in the New Library window and selected in that window to be enabled at startup.

Details

You must set this option by using the New Library window. Enter the name of your libref and select the checkbox labeled Enable at startup to save your librefs. The only way to create librefs that can be autoassigned is through this window. The STARTLIB system option controls whether or not the librefs assigned in this window are assigned at startup.

SUMSIZE= System Option

Specifies a limit on the amount of memory that is available for data summarization procedures when class variables are active

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window
Category: System administration: Memory
PROC OPTIONS GROUP= MEMORY
Syntax

SUMSIZE=n | nK | nM | nG | nT | hexX | MIN | MAX

Syntax Description

n | nK | nM | nG | nT
specifies the amount of memory in terms of 1 (bytes); 1,024 (kilobytes); 1,048,576 (megabytes); 1,073,741,824 (gigabytes); or 1,099,511,627,776 (terabytes). When n=0, the default value, the amount of memory is determined by values of the MEMSIZE option and the REALMEMSIZE option. Valid values for SUMSIZE range from 0 to 2
\((\times \cdot)\) where \(n\) is the data width in bits (32 or 64) of the operating system.

hexX
specifies the amount of memory as a hexadecimal number. You must specify the value beginning with a number (0–9), followed by an X. For example, a value of \texttt{0fffx} specifies 4,095 bytes of memory.

MIN
specifies the minimum amount of memory available.

MAX
specifies the maximum amount of memory available.

Details

The SUMSIZE= system option affects the MEANS, OLAP, REPORT, SUMMARY, SURVEYFREQ, SURVEYLOGISTIC, SURVEYMEANS, and TABULATE procedures. Proper specification of SUMSIZE= can improve procedure performance by restricting the swapping of memory that is controlled by the operating environment.

Generally, the value of the SUMSIZE= system option should be less than the physical memory available to your process. If the procedure you are using needs more memory than you specify, the system creates a temporary utility file.

If the value of SUMSIZE is greater than the values of the MEMSIZE option and the REALMEMSIZE option, SAS uses the values of the MEMSIZE option and REALMEMSIZE option.

See Also

System Options:

“SORTSIZE= System Option” on page 1603

“MEMSIZE System Option” in the documentation for your operating environment.

“REALMEMSIZE System Option” in the documentation for your operating environment.

SYMBOLGEN System Option

Controls whether the results of resolving macro variable references are written to the SAS log
SYNTAXCHECK System Option

Enables syntax check mode for multiple steps in non-interactive or batch SAS sessions

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window
Category: Environment control: Error handling
PROC OPTIONS GROUP= ERRORHANDLING

Syntax
SYNTAXCHECK | NOSYNTAXCHECK

Syntax Description

SYNTAXCHECK
   enables syntax check mode for statements that are submitted within a non-interactive or batch SAS session.

NOSYNTAXCHECK
   does not enable syntax check mode for statements that are submitted within a non-interactive or batch SAS session.

Details
If a syntax or semantic error occurs in a DATA step after the SYNTAXCHECK option is set, then SAS enters syntax check mode, which remains in effect from the point where SAS encountered the error to the end of the code that was submitted. After SAS enters syntax mode, all subsequent DATA step statements and PROC step statements are validated.

While in syntax check mode, only limited processing is performed. For a detailed explanation of syntax check mode, see “Syntax Check Mode” in the section “Error Processing in SAS” in SAS Language Reference: Concepts.

CAUTION:
Place the OPTIONS statement that enables SYNTAXCHECK before the step for which you want it to take effect. If you place the OPTIONS statement inside a step, then SYNTAXCHECK will not take effect until the beginning of the next step. △

CAUTION:
Setting NOSYNTAXCHECK might cause a loss of data. Manipulating and deleting data by using untested code might result in a loss of data if your code contains invalid syntax. Be sure to test code completely before placing it in a production environment. △
NOSYNTAXCHECK enables continuous processing of statements regardless of syntax error conditions.

SYNTAXCHECK is ignored in the SAS windowing environment and in SAS line-mode sessions.

Comparisons

You use the SYNTAXCHECK system option to validate syntax in a non-interactive or a batch SAS session. You use the DMSSYNCHK system option to validate syntax in an interactive session by using the SAS windowing environment.

The ERRORCHECK= option can be set to enable or disable syntax check mode for the LIBNAME statement, the FILENAME statement, the %INCLUDE statement, and the LOCK statement in SAS/SHARE. If you specify the NOSYNTAXCHECK option and the ERRORCHECK=STRICT option, then SAS does not enter syntax check mode when an error occurs.

See Also

System Options:

“DMSSYNCHK System Option” on page 1516
“ERRORCHECK= System Option” on page 1527

“Error Processing in SAS” in the section “Error Processing and Debugging” in SAS Language Reference: Concepts

---

SYS Parm= System Option

Specifies a character string that can be passed to SAS programs

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Environment control: Files

PROC OPTIONS GROUP= ENVFILES

See: SYSPARM= System Option in the documentation for your operating environment.

See also: The SYSPARM System Option in SAS Macro Language: Reference.

---

SYS PRINT FONT= System Option

Sets the font for the current default printer

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Log and procedure output control: Procedure output

PROC OPTIONS GROUP= LISTCONTROL

See: SYS PRINT FONT= System Option in the documentation for your operating environment.
Syntax

SYSPRINTFONT="(face-name" <weight> <style> <character-set> <point-size>
 <NAMED "printer-name " | DEFAULT | ALL)"

Syntax Description

“face-name”
specifies the name of the font face to use for printing. This must be a valid, case-sensitive font face name that matches the name of the font as it is installed on your operating environment.

Requirement: If face-name consists of more than one word, you must be enclose the value in double quotation marks. The quotation marks are stored with the face-name.

Requirement: When you use the SYSPRINTFONT= option with multiple arguments, you must enclose the arguments in parentheses.

weight
specifies the weight of the font, such as BOLD. A list of valid values for your specified printer appears in the SAS: Printer Properties window.
Default: NORMAL

style
specifies the style of the font, such as Italic. A list of valid values for your specified printer appears in the SAS: Printer Properties window.
Default: REGULAR

character-set
specifies the character set to use for printing.
Default: If the font does not support the specified character set, the default character set is used. If the default character set is not supported by the font, the font's default character set is used.
Range: Valid values are listed in the SAS: Printer Properties window, under the Font tab.

point-size
specifies the point size to use for printing. If you omit this argument, SAS uses the default.

Requirement: Point-size must be an integer. It must also be placed after the face-name, weight, style, and character-set.

NAMED “printer-name”
specifies the printer to which these settings apply.
Requirement: The printer-name must exactly match the name shown in the Print Setup dialog box (except that the printer name is not case sensitive).
Requirement: If it is more than one word, the printer-name must be enclosed in double quotation marks. The quotation marks are stored with the printer-name.

DEFAULT
specifies that these settings are applied to the current default printer by default.
ALL
updates the font information for all installed printers.

Details
The SYSPRINTFONT system option sets the font to use when printing to the current
default printer, to another printer specified, or to all printers.

In some cases, you may need to specify the font from a SAS program. In this case,
you may still want to view the SAS: Printer Properties window for allowable names,
styles weights, and sizes for your fonts. For examples of how to apply SYSPRINTFONT
in a SAS program, see “Examples” on page 1612.

If you specified SYSPRINTFONT with DEFAULT or without a keyword and later use
the Print Setup dialog box to change the current default printer, then the font used
with the current default printer will be the font that was specified with
SYSPRINTFONT, if the specified font exists on the printer. If the current printer does
not support the specified font, the printer's default font is used.

The following fonts are widely supported:

- Helvetica
- Times
- Courier
- Symbol.

By specifying one of these fonts in a SAS program, you can usually avoid returning an
error. If that particular font is not supported, a similar-looking font prints in its place.

Note: As an alternative, you can set fonts with the SAS: Printer Properties window,
under the Font tab. From the drop-down menu select File > Print Setup > Properties >
Font. This is fast and easy because you choose your font, style, weight, size, and
character set from a list of options your selected printer supports.

Examples
Specifying a Font to the Default Printer
This example specifies the 12-point Times font on the default printer:

```
options sysprintfont=('times' 12);
```

Specifying a Font to a Named Printer
This example specifies to use Courier on the printer named HP LaserJet IIIb
Postscript. Specify the printer name in the same way that it is specified in the
SAS Print Setup dialog box:

```
options sysprintfont=('courier' named 'hp laserjet 111s, postscript');
```

 TERMINAL System Option

Determines whether SAS evaluates the execution mode and, if needed, resets the option

Valid in: configuration file, SAS invocation

Category: Environment control: Initialization and operation

PROC OPTIONS GROUP= EXECMODES
**Syntax**

TERMINAL | NOTERMINAL

**Syntax Description**

**TERMINAL**
causes SAS to evaluate the execution environment and to set the option to NOTERMINAL if an interactive environment (physical display) is not available. Specify TERMINAL when you use a windowing environment.

**NOTERMINAL**
causes SAS not to evaluate the execution environment. When you specify NOTERMINAL, SAS uses option settings that are associated with the BATCH system option.

**Details**

SAS defaults to the appropriate setting for the TERMINAL system option based on whether the session is invoked in the foreground or the background. If NOTERMINAL is specified, requester windows are not displayed. The TERMINAL option is normally used with the execution modes of windowing mode, interactive line mode, and noninteractive mode.

**See Also**

System Option:
“BATCH System Option” on page 1479

**TERMSTMT= System Option**

Specifies the SAS statements to be executed when the SAS session is terminated

Valid in: configuration file, SAS invocation
Category: Environment control: Initialization and operation
PROC OPTIONS GROUP= EXECMODES

**Syntax**

TERMSTMT='statement(s)'

**Syntax Description**
TERMSTMT= is one or more SAS statements.

Maximum length: 2,048 characters

Operating Environment Information: In some operating system environments there is a limit to the size of the value for TERMSTMT=. To circumvent this limitation, you can use the %INCLUDE statement.

Details

TERMSTMT= is fully supported in batch mode. In interactive modes, TERMSTMT= is executed only when you submit the ENDSAS statement from an editor window to terminate the SAS session. Terminating SAS by any other means in interactive mode results in TERMSTMT= being executed.

An alternate method for specifying TERMSTMT= is to put a %INCLUDE statement at the end of a batch file or submit a %INCLUDE statement before terminating the SAS session in interactive mode.

Comparisons

TERMSTMT= specifies the SAS statements to be executed at SAS termination, and INITSTMT= specifies the SAS statements to be executed at SAS initialization.

See Also

System Option:

“INITSTMT= System Option” on page 1541

Statement:

“%INCLUDE Statement” on page 1218

TEXTURELOC= System Option

Specifies the location of textures and images that are used by ODS styles

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Log and procedure output control: ODS printing

Syntax

TEXTURELOC=location

Syntax Description

location
SAS System Options  △  THREADS System Option  1615

specifies the location of textures and images used by ODS styles. Location can refer either to the physical name of the directory or to a URL reference to the directory.

**Requirement:** If location is not a fileref, then you must enclose the value in quotation marks.

**Restriction:** Only one location is allowed per statement.

**Requirement:** The files in the directory must be in the form of gif, jpeg, or bitmap.

**Requirement:** Location must refer to a directory.

**See Also**


---

**THREADS System Option**

Specifies that SAS use threaded processing if it is available

**Valid in:** configuration file, SAS invocation, OPTIONS statement, SAS System Options window

**Category:** System administration: Performance

**PROC OPTIONS GROUP=** PERFORMANCE

**Syntax**

THREADS | NOTHREADS

**Syntax Description**

**THREADS**

specifies to use threaded processing for SAS applications that support it.

**Interaction** If THREADS is specified either as a SAS system option or in PROC SORT and another program has the input SAS data set open for reading, writing, or updating using the SPD engine, then the procedure might fail and write a subsequent message to the SAS log.

**NOTHREADS**

specifies not to use threaded processing for running SAS applications that support it.

**Interaction:** When you specify NOTHREADS, CPUCOUNT= is ignored unless you specify a procedure option that overrides the NOTHREADS system option.

**Details**

The THREADS system option enables some legacy SAS processes that are thread-enabled to take advantage of multiple CPUs by threading the processing and I/O operations. This achieves a degree of parallelism that generally reduces the real time to completion for a given operation at the possible cost of additional CPU resources. In SAS 9 and SAS 9.1, the thread-enabled processes include
Base SAS engine indexing
Base SAS procedures: SORT, SUMMARY, MEANS, REPORT, TABULATE, and SQL
SAS/STAT procedures: GLM, LOESS, REG, ROBUSTREG.

In some cases, for instance, processing small data sets, SAS might determine to use a single-threaded operation.
Set this option to NOTHREADS to achieve SAS behavior most compatible with releases prior to SAS 9, if you find that threading does not improve performance or if threading might be related to an unexplainable problem. See the specific documentation for each product to determine if it has functionality that is enabled by the THREADS option.

Comparisons

The system option THREADS determines when threaded processing is in effect. The SAS system option CPUCOUNT= suggests how many system CPUs are available for use by thread-enabled SAS procedures.

See Also

System Option:
“CPUCOUNT= System Option” on page 1503
“UTILLOC= System Option” on page 1620

TOOLSMENU System Option

Specifies to include or suppress the Tools menu in windows that display menus

Default: TOOLSMENU
Valid in: configuration file, SAS invocation
Category: Environment control: Display
PROC OPTIONS GROUP= ENVDISPLAY

Syntax

TOOLSMENU | NOTOOLSMENU

Syntax Description

TOOLSMENU
specifies to include the Tools menu in the main menu.

NOTOOLSMENU
specifies to suppress the Tools menu in the main menu.
Details

The Tools pull-down menu is available from all SAS applications. To suppress the Tools pull-down menu, specify NOTOOLSMENU when you invoke SAS.

TOPMARGIN= System Option

Specifies the size of the margin at the top of the page for the ODS printer destination

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Log and procedure output control: ODS printing

PROC OPTIONS GROUP= ODSPRINT

Syntax

TOPMARGIN=margin-size | “margin-size [margin-unit]”

Syntax Description

margin-size
specifies the size of the margin.

Restriction: The bottom margin should be small enough so that the top margin plus the bottom margin is less than the height of the paper.

Interactions: Changing the value of this option may result in changes to the value of the PAGESIZE= system option.

[margin-unit]
specifies the units for margin-size. The margin-unit can be in for inches or cm for centimeters.

Default: inches

Requirement: When you specify margin-unit, enclose the entire option value in double quotation marks.

Details

Note that all margins have a minimum that is dependent on the printer and the paper size.

Operating Environment Information: Most SAS system options are initialized with default settings when SAS is invoked. However, the default settings and/or option values for some SAS system options may vary both by operating environment and by site. For details, see the SAS documentation for your operating environment.

For additional information on declaring an ODS printer destination, see the ODS statements in SAS Output Delivery System: User's Guide.
See Also

System Options:
   “BOTTOMMARGIN= System Option” on page 1481
   “LEFTMARGIN= System Option” on page 1544
   “RIGHTMARGIN= System Option” on page 1592

TRAINLOC= System Option

Specifies the base location of SAS online training courses

Valid in:   configuration file, SAS invocation
Category:   Environment control: Files
PROC OPTIONS GROUP= ENVFILES

Syntax

TRAINLOC="base-URL"

Syntax Description

base-URL
   specifies the address where the SAS online training courses are located.

Details

The TRAINLOC= system option specifies the base location (typically a URL) of SAS online training courses. These online training courses are typically accessed from an intranet server or a local CD-ROM.

Examples

Some examples of the base-URL are:
   "file://e:\onlintut"
   "http://server.abc.com/SAS/sastrain"

TRANTAB= System Option

Specifies the translation tables that are used by various parts of SAS

Valid in:   configuration file, SAS invocation, OPTIONS statement, SAS System Options window
Category:   Environment control: Language control
PROC OPTIONS GROUP= LANGUAGECONTROL
SAS System Options △ USER= System Option

---


---

**UNIVERSALPRINT System Option**

Specifies whether to enable Universal Printing services

**Valid in:** configuration file, SAS invocation

**Category:** Log and procedure output control: ODS printing

**PROC OPTIONS GROUP=** ODSPRINT

**Syntax**

UNIVERSALPRINT | NOUNIVERSALPRINT

**Syntax Description**

**UNIVERSALPRINT**

routes all printing through the Universal Print services.

**Alias:** UPRINT

**NOUNIVERSALPRINT**

disables printing through the Universal Print services.

**Alias:** NOUPRINT

**Details**

Universal Printing services provides interactive and batch printing capabilities to SAS applications and procedures. The ODS PRINTER destination uses Universal Print services whether or not the UNIVERSALPRINT option is enabled.

**See Also**

“Printing with SAS” in SAS Language Reference: Concepts

---

**USER= System Option**

Specifies the default permanent SAS data library

**Valid in:** configuration file, SAS invocation, OPTIONS statement, SAS System Options window

**Category:** Environment control: Files

**PROC OPTIONS GROUP=** ENVFILES
See: USER= System Option in the documentation for your operating environment.

Syntax
USER= library-specification

Syntax Description

library-specification specifies the libref or physical name of a SAS data library.

Details
If this option is specified, you can use one-level names to reference permanent SAS files in SAS statements. However, if USER=WORK is specified, SAS assumes that files referenced with one-level names refer to temporary work files.

Operating Environment Information: The syntax that is shown here applies to the OPTIONS statement. On the command line or in a configuration file, the syntax is specific to your operating environment. For details, see the SAS documentation for your operating environment. △

UTILLOC= System Option

Specifies one or more file system locations in which applications can store utility files

Valid in: configuration file and SAS invocation
Category: Files: SAS Files
PROC OPTIONS GROUP= SASFILES
See: UTILLOC= System Option in the documentation for your operating environment.

Syntax
UTILLOC= WORK | location | (location-1... location-n)

Syntax Description

WORK specifies that SAS create utility files in the same directory as the WORK library.

location specifies the location of an existing directory for utility files that are created by applications. Enclose location in single or double quotation marks when the location contains spaces.

Operating Environment Information: On z/OS each location is a list of DCB and SMS options to be used when creating utility files. △
(location-1 ... location-n)
specifies a list of existing directories that can be accessed in parallel for utility files that are created by applications. A single utility file cannot span locations. Enclose a location in single or double quotation marks when the location contains spaces.

Operating Environment Information: On z/OS, each location is a list of DCB and SMS options to be used when creating utility files. △

Requirement: If you have more than one location, then you must enclose the list of locations in parentheses.

Details
In SAS 9 and 9.1, thread-enabled SAS applications are able to create temporary utility files that can be accessed in parallel by separate threads.

See Also
System Option:
“CPUCOUNT= System Option” on page 1503
“THREADS System Option” on page 1615

UUIDCOUNT= System Option

Specifies the number of UUIDs to acquire each time the UUID Generator Daemon is used

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window
Category: Environment control: Files
PROC OPTIONS GROUP= ENVFILES

Syntax
UUIDCOUNT= n | MIN | MAX

Syntax Description

n specifies the number of UUIDs to acquire.
Range: 0–1000

Note: Zero indicates that the UUID Generator Daemon is not required. △

Default: 100
MIN | MAX
MIN specifies that the number of UUIDs to acquire is zero, indicating that the UUID Generator Daemon is not required.

MAX specifies that 1000 UUIDs at a time should be acquired from the UUID Generator Daemon.

Details
If a SAS application will generate a large number of UUIDs, this value can be adjusted at any time during a SAS session to reduce the number of times that the SAS session would have to contact the SAS UUID Generator Daemon.

See Also
System Option:

“UUIDGENDHOST= System Option” on page 1622

Function:

“UUIDGEN Function” on page 871

“Universal Unique Identifiers and the Object Spawner” in SAS Language Reference: Concepts

UUIDGENDHOST= System Option

Identifies the host and the port of the UUID Generator Daemon

Valid in: configuration file, SAS invocation

Category: Environment control: Files

PROC OPTIONS GROUP= ENVFILES

Syntax

UUIDGENDHOST= 'host-string'

Syntax Description

'host-string'
is either of the form hostname:port or an LDAP URL. The value must be in one string. Enclose a LDAP URL string with quotation marks.

Details
When SAS executes under operating environments other than Windows NT, SAS does not guarantee that all UUIDs are unique. Use the SAS UUID Generator Daemon (UUIDGEN) to ensure unique UUIDs on operating environments other than Windows NT.
Examples

- Specifying hostname:port as the 'host-string':
  ```
sas -UUIDGENDHOST 'myhost.com:5306'
  or
  sas UUIDGENDHOST= 'myhost.com:5306'
  ```

- Specifying an LDAP URL as the 'host-string':
  ```
  "ldap://ldaphost/sasSpawner-distinguished-name"
  ```

- A more detailed example of an LDAP URL as the 'host-string':
  ```
  "ldap://ldaphost/sasSpawnercn=UUIDGEND,sascomponent=sasServer, cn=ABC,o=ABC Inc,c=US"
  ```

- Specifying your binddn and password, if your LDAP server is secure:
  ```
  "ldap://ldaphost/sasSpawner-distinguished-name
  bindname=binddn,password=bind-password"
  ```

- An example with a bindname value and a password value:
  ```
  "ldap://ldaphost/
  sasSpawnercn=UUIDGEND,sascomponent=sasServer, cn=ABC,o=ABC Inc,c=US
  bindname=cn=me%2co=ABC Inc %2cc=US,
  password=itsme"
  ```

  **Note:** When specifying your bindname and password, commas that are a part of your bindname and your password must be replaced with the string "%2c". In the previous example, the bindname is as follows:
  ```
  cn=me,o=ABC Inc,c=US
  ```

See Also

- System Option:
  “UUIDCOUNT= System Option” on page 1621

- Function:
  “UUIDGEN Function” on page 871

---

**V6CREATEUPDATE= System Option**

Controls or monitors the creation of new Version 6 data sets or the updating of existing Version 6 data sets.

**Valid in:** configuration file, SAS invocation

**Category:** Environment control: Files

**PROC OPTIONS GROUP=** SASFILES

**Syntax**

V6CREATEUPDATE = ERROR | NOTE | WARNING | IGNORE
### Syntax Description

**ERROR**
specifies that an ERROR is written to the SAS log when the V6 engine is used to open a SAS data set for creation or update. The attempt to create or update a SAS data set in Version 6 format will fail. Reading Version 6 data sets will not generate an error.

**NOTE**
specifies that a NOTE is written to the SAS log when the V6 engine is used; all other processing occurs normally.

**WARNING**
specifies that a WARNING is written to the SAS log when the V6 engine is used; all other processing occurs normally.

**IGNORE**
disables the V6CREATEUPDATE= system option. Nothing is written to the SAS log when the V6 engine is used.

---

**VALIDFMTNAME= System Option**

Controls the length for the names of formats and informats that can be used when creating new SAS data sets and format catalogs

**Default:** LONG

**Valid in:** configuration file, SAS invocation, OPTIONS statement, SAS System Options window

**Category:** Files: SAS Files

**PROC OPTIONS GROUP= SASFILES**

---

**Syntax**

VALIDFMTNAME=LONG | FAIL | WARN

---

**Syntax Description**

**LONG**
specifies that format and informat names can be up to 32 alphanumeric characters. This is the default.

**FAIL**
specifies that using a format or informat name that is longer than eight characters results in an error message.

**Tip:** Specify this setting for using formats and informats that are valid in both SAS System 9 and previous releases of SAS.

**Interaction:** If you explicitly specify the V7 or V8 Base SAS engine, such as in a LIBNAME statement, then SAS automatically uses the VALIDFMTNAME=FAIL behavior for data sets that are associated with those engines.
WARN

is the same as FAIL, except that if a format or informat name exceeds eight characters, a warning message is displayed to remind you that the format or informat cannot be used with releases prior to SAS System 9.

Details

SAS System 9 enables you to define format and informat names up to 32 characters. Previous releases were limited to eight characters. The VALIDFMTNAME= system option applies to format and informat names in both data sets and format catalogs. VALIDFMTNAME= does not control the length of format and informat names. It only controls the length of format and informat names that you associate with variables when you create a SAS data set.

If a SAS data set has a variable with a long format or informat name, which means that a release prior to SAS System 9 cannot read it, then you can remove the long name so that the data set can be accessed by an earlier release. However, in order to retain the format attribute of the variable, an identical format with a short name would have to be applied to the variable.

Note: After you create a format or informat using a name that is longer than eight characters, if you rename it using eight or fewer characters, a release prior to SAS System 9 cannot use the format or informat. You must recreate the format or informat using the shorter name.

See Also

For more information about SAS names, see “Names in the SAS Language” Names in the SAS Language in SAS Language Reference: Concepts.

For information about defining formats and informats, see “The FORMAT Procedure” in Base SAS Procedures Guide.


VALIDVARNAME= System Option

Controls the type of SAS variable names that can be created and processed during a SAS session

Default: V7

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Files: SAS Files

PROC OPTIONS GROUP= SASFILES

Syntax

VALIDVARNAME=V7 | UPCASE | ANY

Syntax Description
V7
indicates that a variable name
- can be up to 32 characters in length.
- must begin with a letter of the Latin alphabet (A - Z, a - z) or the underscore character. Subsequent characters can be letters of the Latin alphabet, numerals, or underscores.
- cannot contain blanks.
- cannot contain special characters except for the underscore.
- can contain mixed-case letters. SAS stores and writes the variable name in the same case that is used in the first reference to the variable. However, when SAS processes a variable name, SAS internally converts it to uppercase. You cannot, therefore, use the same variable name with a different combination of uppercase and lowercase letters to represent different variables. For example, cat, Cat, and CAT all represent the same variable.
- cannot be assigned the names of special SAS automatic variables (such as _N_ and _ERROR_) or variable list names (such as _NUMERIC_, _CHARACTER_, and _ALL_).

UPCASE
indicates that the variable name follows the same rules as V7, except that the variable name is uppercase, as in earlier versions of SAS.

ANY
indicates that a variable name
- can be up to 32 characters in length.
- can begin with or contain any characters, including blanks.

CAUTION: Available for Base SAS procedures and SAS/STAT procedure only. Any other use of this option is considered experimental and might cause undetected errors. △

Note: If you use any characters other than the ones that are valid when the VALIDVARNAME system option is set to V7 (letters of the Latin alphabet, numerals, or underscores), then you must express the variable name as a name literal and you must set VALIDVARNAME=ANY. See “SAS Name Literals” and “Avoiding Errors When Using Name Literals” in SAS Language Reference: Concepts. △

- can contain mixed-case letters. SAS stores and writes the variable name in the same case that is used in the first reference to the variable. However, when SAS processes a variable name, SAS internally converts it to uppercase. You cannot, therefore, use the same variable name with a different combination of uppercase and lowercase letters to represent different variables. For example, cat, Cat, and CAT all represent the same variable.

See Also

“Rules for Words and Names in the SAS Language” in SAS Language Reference: Concepts
**VIEWMENU System Option**

Specifies to include or suppress the View menu in windows that display menus

Default: VIEWMENU

Valid in: configuration file, SAS invocation

Category: Environment control: Display

PROC OPTIONS GROUP= ENVDISPLAY

**Syntax**

VIEWMENU | NOVIEWMENU

**Syntax Description**

**VIEWMENU**

specifies to include the View menu in the main menu.

**NOVIEWMENU**

specifies to suppress the View menu in the main menu.

**Details**

The View pull-down menu is available from all SAS applications. To suppress the View pull-down menu, specify NOVIEWMENU when you invoke SAS.

---

**VNFERR System Option**

Controls how SAS responds when a _NULL_ data set is used

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Environment control: Error handling

PROC OPTIONS GROUP= ERRORHANDLING

**Syntax**

VNFERR | NOVNFERR

**Syntax Description**

**VNFERR**

specifies that SAS either issue a warning and error message, or set _ERROR_=1 and stop processing, or both if the error is severe enough to interrupt processing.
NOVFERR
specifies that SAS issue a warning for a variable that is not found, but not set _ERROR_=1 or stop processing.

Details
The VNFERR system option specifies whether SAS sets the error flag (_ERROR_=1) for a missing variable when a _NULL_ data set (or a data set that is bypassed by the operating environment control language) is used in a MERGE statement within a DATA step.

Comparisons
- VNFERR is similar to the BYERR system option, which issues an error message and stops processing if the SORT procedure attempts to sort a _NULL_ data set.
- VNFERR is similar to the DSNFERR system option, which generates an error message when a SAS data set is not found.

See Also
System Options:
“BYERR System Option” on page 1485
“DSNFERR System Option” on page 1517

---

WORK= System Option

Specifies the WORK data library

Valid in: configuration file, SAS invocation
Category: Environment control: Files
PROC OPTIONS GROUP= ENVFILES
See: WORK= System Option in the documentation for your operating environment.

Syntax

WORK=library-specification

Syntax Description

library-specification
specifies the libref or physical name of the storage space where all data sets with one-level names are stored. This library must exist.

Operating Environment Information: A valid library specification and its syntax are specific to your operating environment. On the command line or in a configuration file, the syntax is specific to your operating environment. For details, see the SAS documentation for your operating environment.
Details
This library is deleted at the end of your SAS session by default. To prevent the files from being deleted, specify the NOWORKTERM system option.

See Also

System Option:
“WORKTERM System Option” on page 1630

WORKINIT System Option

Initializes the WORK data library
Valid in: configuration file, SAS invocation
Category: Environment control: Files
PROC OPTIONS GROUP= ENVFILES

Syntax
WORKINIT | NOWORKINIT

Syntax Description

WORKINIT
erases files that exist from a previous SAS session in an existing WORK library at SAS invocation.

NOWORKINIT
does not erase files from the WORK library at SAS invocation.

Comparisons
The WORKINIT system option initializes the WORK data library and erases all files from a previous SAS session at SAS invocation. The WORKTERM system option controls whether or not SAS erases WORK files at the end of a SAS session.

See Also

System Option:
“WORKTERM System Option” on page 1630

Operating Environment Information: WORKINIT has behavior and functions specific to the UNIX operating environment. For details, see the SAS documentation for the UNIX operating environment. △
WORKTERM System Option

Controls whether SAS erases WORK files at the termination of a SAS session

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Environment control: Files

PROC OPTIONS GROUP= ENVFILES

Syntax

WORKTERM | NOWORKTERM

Syntax Description

WORKTERM

erases the WORK files at the termination of a SAS session.

NOWORKTERM

does not erase the WORK files.

Details

Although NOWORKTERM prevents the WORK data sets from being deleted, it has no effect on initialization of the WORK library by SAS. SAS normally initializes the WORK library at the start of each session, which effectively destroys any pre-existing information.

Comparisons

Use the NOWORKINIT system option to prevent SAS from erasing existing WORK files on invocation. Use the NOWORKTERM system option to prevent SAS from erasing existing WORK files on termination.

See Also

System Option:

“WORKINIT System Option” on page 1629

YEARCUTOFF= System Option

Specifies the first year of a 100-year span that will be used by date informats and functions to read a two-digit year

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window
Category: Input control: Data Processing
PROC OPTIONS GROUP= INPUTCONTROL

Syntax
YEARCUTOFF= nnnn | nnnnn

Syntax Description

nnnn | nnnnn
    specifies the first year of the 100-year span.
    Range: 1582–19900
    Default: 1920

Details
The YEARCUTOFF= value is the default that is used by various date and datetime informats and functions.

If the default value of nnn (1920) is in effect, the 100-year span begins with 1920 and ends with 2019. Therefore, any informat or function that uses a two-digit year value that ranges from 20 to 99 assumes a prefix of 19. For example, the value 92 refers to the year 1992.

The value that you specify in YEARCUTOFF= can result in a range of years that span two centuries. For example, if you specify YEARCUTOFF=1950, any two-digit value between 50 and 99 inclusive refers to the first half of the 100-year span, which is in the 1900s. Any two-digit value between 00 and 49, inclusive, refers to the second half of the 100-year span, which is in the 2000s. The following figure illustrates the relationship between the 100-year span and the two centuries if YEARCUTOFF=1950.

![Figure 8.1 A 100-Year Span with Values in Two Centuries](image)

Note: YEARCUTOFF= has no effect on existing SAS dates or dates that are read from input data that include a four-digit year, except those with leading zeroes. For example, 0076 with yearcutoff=1990 indicates 1976.

Operating Environment Information: The syntax that is shown here applies to the OPTIONS statement. On the command line or in a configuration file, the syntax is specific to your operating environment. For details, see the SAS documentation for your operating environment.

See Also
Appendixes

Appendix 1. ........ DATA Step Object Attributes and Methods 1635
Appendix 2. ........ DATA Step Debugger 1663
Appendix 3. ........ SAS Utility Macro 1695
Appendix 4. ........ Recommended Reading 1699
APPENDIX

1

DATA Step Object Attributes and Methods

The DATA Step Component Object Interface

The DATA step component object interface enables you to create and manipulate predefined component objects in a DATA step. Component objects are data elements that consist of attributes and methods. Attributes are the properties that specify the information that is associated with an object. An example is size. Methods define the operations that an object can perform.

SAS provides two predefined component objects for use in a DATA step: the hash object and the hash iterator object. These objects enable you to quickly and efficiently store, search, and retrieve data based on lookup keys. For more information about these objects, see “Using DATA Step Component Objects” in SAS Language Reference: Concepts.

You use the DECLARE and _NEW_ statements to declare and create a component object. You use the DATA step object dot notation to access the component object’s attributes and methods.
Dot Notation and DATA Step Component Objects

Definition

Dot notation provides a shortcut for invoking methods and for setting and querying attribute values. Using dot notation makes your SAS programs easier to read.

To use dot notation with a DATA step component object, you must declare and instantiate the component object by using the DECLARE and _NEW_ statements. For more information about creating DATA step component objects, see “Using DATA Step Component Objects” in SAS Language Reference: Concepts.

Syntax

The syntax for dot notation is as follows:

object.attribute

or

object.method(<argument_tag-1: value-1<, ...argument_tag-n: value-n>>);

Where

object
  specifies the variable name for the DATA step component object

attribute
  specifies an object attribute to assign or query.
  When you set an attribute for an object, the code takes this form:
  object.attribute = value;
  When you query an object attribute, the code takes this form:
  value = object.attribute;

method
  specifies the name of the method to invoke.

argument_tag
  identifies the arguments that are passed to the method. Enclose the argument tag in parentheses. The parentheses are required whether or not the method contains argument tags.
  All DATA step component object methods take this form:
  return_code = object.method(<argument_tag-1: value-1<, ...argument_tag-n: value-n>>);

  The return code indicates method success or failure. A return code of zero indicates success; a non-zero value indicates failure. If you do not supply a return code variable for the method call and the method fails, an appropriate error message will be printed to the log.
Dictionary

Hash Object Attributes and Methods

The hash object contains the following attributes and methods.

Attribute

“NUM_ITEMS Attribute” on page 1652

Methods

“ADD Method” on page 1637
“CHECK Method” on page 1639
“DEFINEDATA Method” on page 1641
“DEFINEDONE Method” on page 1643
“DEFINEKEY Method” on page 1644
“DELETE Method” on page 1645
“FIND Method” on page 1646
“OUTPUT Method” on page 1653
“REMOVE Method” on page 1656
“REPLACE Method” on page 1659

Hash Iterator Object Methods

“FIRST Method” on page 1648
“LAST Method” on page 1650
“NEXT Method” on page 1651
“PREV Method” on page 1656

ADD Method

Adds the specified data that is associated with the given key to the hash object.

Applies to: Hash object

Syntax

rc=object.ADD(<KEY: keyvalue-1, ..., KEY: keyvalue-n, DATA: datavalue-1, ..., DATA: datavalue-n>);
## Arguments

**rc**

specifies whether the method succeeded or failed.

A return code of zero indicates success; a non-zero value indicates failure. If you do not supply a return code variable for the method call and the method fails, then an appropriate error message is written to the log.

**object**

specifies the name of the hash object.

**KEY: keyvalue**

specifies the key value whose type must match the corresponding key variable that is specified in a DEFINEKEY method call.

The number of “KEY: keyvalue” pairs depends on the number of key variables that you define by using the DEFINEKEY method.

**DATA: datavalue**

specifies the data value whose type must match the corresponding data variable that is specified in a DEFINEDATA method call.

The number of “DATA: datavalue” pairs depends on the number of data variables that you define by using the DEFINEDATA method.

## Details

You can use the ADD method in one of two ways to store data in a hash object.

You can define the key and data item, and then use the ADD method as shown in the following code:

```bash
data _null_
length k $8;
length d $12;

/* Declare hash object and key and data variable names*/
if _N_ = 1 then do;
  declare hash h(hashexp: 4);
  rc = h.defineKey('k');
  rc = h.defineData('d');
  rc = h.defineDone();
end;

/* Define constant key and data values */
k = 'Joyce';
d = 'Ulysses';
/* Add key and data values to hash object */
rc = h.add();
run;
```

Alternatively, you can use a shortcut and specify the key and data directly in the ADD method call as shown in the following code:

```bash
data _null_
length k $8;
length d $12;

/* Define hash object and key and data variable names*/
if _N_ = 1 then do;
```
declare hash h(hashexp: 4);
rc = h.defineKey('k');
rc = h.defineData('d');
rc = h.defineDone();
/* avoid uninitialized variable notes */
call missing(k, d);
end;

/* Define constant key and data values and add to hash object */
rc = h.add(key: 'Joyce', data: 'Ulysses');
run;

Note: If you add a key that is already in the hash object, then the ADD method will return a non-zero value to indicate that the key is already in the hash object. Use the REPLACE method to replace the data that is associated with the specified key with new data.

See Also

Statements:
“DEFINEDATA Method” on page 1641
“DEFINEKEY Method” on page 1644
“Storing and Retrieving Data” in SAS Language Reference: Concepts

CHECK Method

Checks whether the specified key is stored in the hash object.

Applies to: Hash object

Syntax

rc=object.CHECK(<KEY: keyvalue-1,..., KEY: keyvalue-n>);

Arguments

rc
specifies whether the method succeeded or failed.
A return code of zero indicates success; a non-zero value indicates failure. If you do not supply a return code variable for the method call and the method fails, then an appropriate error message is written to the log.

object
specifies the name of the hash object.

KEY: keyvalue
specifies the key value whose type must match the corresponding key variable that is specified in a DEFINEKEY method call.
The number of “KEY: keyvalue” pairs depends on the number of key variables that you define by using the DEFINEKEY method.
Details

You can use the CHECK method in one of two ways to store data in a hash object.

You can specify the key, and then use the CHECK method as shown in the following code:

```plaintext
data _null_
length k $8;
length d $12;
/* Declare hash object and key and data variable names*/
if _N_ = 1 then do;
   declare hash h(hashexp: 4);
   rc = h.defineKey('k');
   rc = h.defineData('d');
   rc = h.defineDone();
   /* avoid uninitialized variable notes */
   call missing(k, d);
end;
/* Define constant key and data values and add to hash object */
rc = h.add(key: 'Joyce', data: 'Ulysses');
/* Verify that JOYCE key is in hash object */
k = 'Joyce';
rc = h.check();
if (rc = 0) then
   put 'Key is in the hash object.';
run;
```

Alternatively, you can use a shortcut and specify the key directly in the CHECK method call as shown in the following code:

```plaintext
data _null_
length k $8;
length d $12;
/* Declare hash object and key and data variable names*/
if _N_ = 1 then do;
   declare hash h(hashexp: 4);
   rc = h.defineKey('k');
   rc = h.defineData('d');
   rc = h.defineDone();
   /* avoid uninitialized variable notes */
   call missing(k, d);
end;
/* Define constant key and data values and add to hash object */
rc = h.add(key: 'Joyce', data: 'Ulysses');
/* Verify that JOYCE key is in hash object */
rc = h.check(key: 'Joyce');
if (rc = 0) then
   put 'Key is in the hash object.';
run;
```
DATA Step Object Attributes and Methods  △  DEFINEDATA Method  1641

Comparison

The CHECK method only returns a value that indicates whether the key is in the hash object. The data variable that is associated with the key is not updated. The FIND method also returns a value that indicates whether the key is in the hash object. However, if the key is in the hash object, then the FIND method also sets the data variable to the value of the data item so that it is available for use after the method call.

See Also

Methods:
“FIND Method” on page 1646
“DEFINEKEY Method” on page 1644

DEFINEDATA Method

Defines data, associated with the specified data variables, to be stored in the hash object

Applies to: Hash object

Syntax

1. `rc=object.DEFINEDATA('datavarname-1',...,'datavarname-n');`
2. `rc=object.DEFINEDATA(ALL: 'YES' | "YES");`

Arguments

`rc`

specifies whether the method succeeded or failed.

A return code of zero indicates success; a non-zero value indicates failure. If you do not supply a return code variable for the method call and the method fails, then an appropriate error message is written to the log.

`object`

specifies the name of the hash object.

`'datavarname'`

specifies the name of the data variable.

The data variable name can also be enclosed in double quotation marks.

`ALL: 'YES' | "YES"`

specifies all the data variables as data when the data set is loaded in the object constructor.

If the `dataset` argument tag is used in the DECLARE or `_NEW_` statement to automatically load a data set, then you can define all the data set variables as data by using the `ALL: 'YES'` option.

Note: If you use the shortcut notation for the ADD or REPLACE method (for example, `h.add(key: 99, data: 'apple', data: 'orange')`), then you should specify the data in the same order as it exists in the data set. △

Note: The hash object does not assign values to key variables (for example, `h.find(key: 'abc')`), and the SAS compiler cannot detect the implicit key and data
variable assignments that are performed by the hash object and the hash iterator. Therefore, if no explicit assignment to a key or data variable appears in the program, then SAS will issue a note stating that the variable is uninitialized. To avoid receiving these notes, you can perform one of the following actions:

- Set the NONOTES system option.
- Provide an initial assignment statement (typically to a missing value) for each key and data variable.
- Use the CALL MISSING routine with all the key and data variables as parameters. Here is an example:

```sas
length d $20;
length k $20;
if _N_ = 1 then do;
    declare hash h(hashexp: 4);
    rc = h.defineKey('k');
    rc = h.defineData('d');
    rc = h.defineDone();
    call missing(k, d);
end;
```

### Details

The hash object works by storing and retrieving data based on lookup keys. The keys and data are DATA step variables, which you use to initialize the hash object by using dot notation method calls. You define a key by passing the key variable name to the DEFINEKEY method. You define data by passing the data variable name to the DEFINEDATA method. When you have defined all key and data variables, you must call the DEFINEDONE method to complete initialization of the hash object. Keys and data consist of any number of character or numeric DATA step variables.

For detailed information about how to use the DEFINEDATA method, see “Defining Keys and Data” in *SAS Language Reference: Concepts*.

### Example

The following example creates a hash object and defines the key and data variables:

```sas
data _null_; 
    length d $20; 
    length k $20; 
    /* Declare the hash object and key and data variables */ 
    if _N_ = 1 then do; 
        declare hash h(hashexp: 4); 
        rc = h.defineKey('k'); 
        rc = h.defineData('d'); 
        rc = h.defineDone(); 
        /* avoid uninitialized variable notes */ 
        call missing(k, d); 
    end; 
    run;
```
See Also

Methods:

“DEFINEDONE Method” on page 1643
“DEFINEKEY Method” on page 1644

Statements:

“DECLARE Statement” on page 1135
“_NEW_ Statement” on page 1327

“Defining Keys and Data” in SAS Language Reference: Concepts

DEFINEDONE Method

Indicates that all key and data definitions are complete.

Applies to: Hash object

Syntax

rc = object.DEFINEDONE( );

Arguments

rc
specifies whether the method succeeded or failed.

A return code of zero indicates success; a non-zero value indicates failure. If you do not supply a return code variable for the method call and the method fails, then an appropriate error message is written to the log.

object
specifies the name of the hash object.

Details

The hash object works by storing and retrieving data based on lookup keys. The keys and data are DATA step variables, which you use to initialize the hash object by using dot notation method calls. You define a key by passing the key variable name to the DEFINEKEY method. You define data by passing the data variable name to the DEFINEDATA method. When you have defined all key and data variables, you must call the DEFINEDONE method to complete initialization of the hash object. Keys and data consist of any number of character or numeric DATA step variables.

For detailed information about how to use the DEFINEDONE method, see “Defining Keys and Data” in SAS Language Reference: Concepts.

See Also

Methods:

“DEFINEDATA Method” on page 1641
“DEFINEKEY Method” on page 1644
“Defining Keys and Data” in *SAS Language Reference: Concepts*

**DEFINEKEY Method**

**Defines key variables for the hash object**

**Applies to:** Hash object

---

**Syntax**

1. \( rc = \text{object}.\text{DEFINEKEY}('keyvarname-1'..., 'keyvarname-n'); \)
2. \( rc = \text{object}.\text{DEFINEKEY}('\text{ALL}: \text{"YES" | "YES"}); \)

**Arguments**

- \( rc \)
  - specifies whether the method succeeded or failed.
  - A return code of zero indicates success; a non-zero value indicates failure. If you do not supply a return code variable for the method call and the method fails, then an appropriate error message is written to the log.

- \( \text{object} \)
  - specifies the name of the hash object.

- 'keyvarname'
  - specifies the name of the key variable.
  - The key variable name can also be enclosed in double quotation marks.

- ALL: 'YES' | "YES"
  - specifies all the data variables as keys when the data set is loaded in the object constructor.
  - If you use the *dataset* argument tag in the DECLARE or _NEW_ statement to automatically load a data set, then you can define all the key variables by using the ALL: 'YES' option.

  *Note:* If you use the shortcut notation for the ADD, CHECK, FIND, REMOVE, or REPLACE methods (for example, \( h.\text{add(key: 99, data: 'apple', data: 'orange')} \)), then you should specify the keys and data in the same order as they exist in the data set.

**Details**

The hash object works by storing and retrieving data based on lookup keys. The keys and data are DATA step variables, which you use to initialize the hash object by using dot notation method calls. You define a key by passing the key variable name to the DEFINEKEY method. You define data by passing the data variable name to the DEFINEDATA method. When you have defined all key and data variables, you must call the DEFINEDONE method to complete initialization of the hash object. Keys and data consist of any number of character or numeric DATA step variables.

For detailed information about how to use the DEFINEKEY method, see “Defining Keys and Data” in *SAS Language Reference: Concepts*.

*Note:* The hash object does not assign values to key variables (for example, \( h.\text{find(key: 'abc')} \)), and the SAS compiler cannot detect the implicit key and data
variable assignments done by the hash object and the hash iterator. Therefore, if no explicit assignment to a key or data variable appears in the program, SAS will issue a note stating that the variable is uninitialized. To avoid receiving these notes, you can perform one of the following actions:

- Set the NONOTES system option.
- Provide an initial assignment statement (typically to a missing value) for each key and data variable.
- Use the CALL MISSING routine with all the key and data variables as parameters. Here is an example:

```sas
length d $20;
length k $20;

if _N_ = 1 then do;
    declare hash h(hashexp: 4);
    rc = h.defineKey('k');
    rc = h.defineData('d');
    rc = h.defineDone();
    call missing(k, d);
end;
```

See Also

Methods:

- “DEFINEDATA Method” on page 1641
- “DEFINEDONE Method” on page 1643

Statements:

- “DECLARE Statement” on page 1135
- “_NEW_ Statement” on page 1327
- “Defining Keys and Data” in SAS Language Reference: Concepts

DELETESMethod

**DELETESMethod**

**Deletes the hash object**

**Applies to:** Hash object

**Syntax**

```sas
rc=object.DELETE();
```

**Arguments**

- `rc` specifies whether the method succeeded or failed.
A return code of zero indicates success; a non-zero value indicates failure. If you do not supply a return code variable for the method call and the method fails, then an appropriate error message is printed to the log.

`object` specifies the name of the hash object.

**Details**

When you no longer need the hash object, delete it by using the `DELETE` method. If you attempt to use a hash object after you delete it, you will receive an error in the log.

---

**FIND Method**

Determines whether the specified key is stored in the hash object.

**Applies to:** Hash object

---

**Syntax**

\[
rc = object.FIND(<KEY: keyvalue–1,..., KEY: keyvalue–n>);
\]

**Arguments**

- \( rc \) specifies whether the method succeeded or failed.
  - A return code of zero indicates success; a non-zero value indicates failure. If you do not supply a return code variable for the method call and the method fails, then an appropriate error message is written to the log.

- \( object \) specifies the name of the hash object.

- \( KEY: keyvalue \) specifies the key value whose type must match the corresponding key variable that is specified in a `DEFINEKEY` method call.
  - The number of “\( KEY: keyvalue \)” pairs depends on the number of key variables that you define by using the `DEFINEKEY` method.

**Details**

You can use the `FIND` method in one of two ways to find data in a hash object.

You can specify the key, and then use the `FIND` method as shown in the following code:

```plaintext
data _null_;
    length k $8;
    length d $12;

    /* Declare hash object and key and data variables */
    if _N_ = 1 then do;
        declare hash h(hashexp: 4);
        rc = h.defineKey('k');
```
DATA Step Object Attributes and Methods

FIND Method

rc = h.defineData('d');
rc = h.defineDone();
/* avoid uninitialized variable notes */
call missing(k, d);
end;

/* Define constant key and data values */
rc = h.add(key: ‘Joyce’, data: ‘Ulysses’);

/* Find the key JOYCE */
k = ‘Joyce’;
rc = h.find();
if (rc = 0) then
    put ‘Key is in the hash object.’;
run;

Alternatively, you can use a shortcut and specify the key directly in the FIND method call as shown in the following code:

data _null_
length k $8;
length d $12;

/* Declare hash object and key and data variables */
if _N_ = 1 then do;
    declare hash h(hashexp: 4);
    rc = h.defineKey('k');
    rc = h.defineData('d');
    rc = h.defineDone();
    /* avoid uninitialized variable notes */
    call missing(k, d);
end;

/* Define constant key and data values */
rc = h.add(key: ‘Joyce’, data: ‘Ulysses’);

/* Find the key JOYCE */
rc = h.find(key: ‘Joyce’);
if (rc = 0) then
    put ‘Key is in the hash object.’;
run;

Comparison

The FIND method returns a value that indicates whether the key is in the hash object. If the key is in the hash object, then the FIND method also sets the data variable to the value of the data item so that it is available for use after the method call. The CHECK method only returns a value that indicates whether the key is in the hash object. The data variable is not updated.

Example

The following example creates a hash object. Two data values are added. The FIND method is used to find a key in the hash object. The data value is returned to the data set variable that is associated with the key.
data _null_; length k $8; length d $12; /* Declare hash object and key and data variable names */ if _N_ = 1 then do; declare hash h(hashexp: 4); rc = h.defineKey('k'); rc = h.defineData('d'); /* avoid uninitialized variable notes */ call missing(k, d); rc = h.defineDone(); end; /* Define constant key and data values and add to hash object */ rc = h.add(key: 'Joyce', data: 'Ulysses'); rc = h.add(key: 'Homer', data: 'Odyssey'); /* Verify that key JOYCE is in hash object and */ /* return its data value to the data set variable D */ rc = h.find(key: 'Joyce'); put d=; run; d=Ulysses is written to the SAS log.

See Also

Methods:

“CHECK Method” on page 1639
“DEFINEKEY Method” on page 1644
“Storing and Retrieving Data” in SAS Language Reference: Concepts

---

**FIRST Method**

Returns the first value in the underlying hash object.

**Applies to:** Hash iterator object

**Syntax**

```plaintext
rc=object.FIRST();
```

**Arguments**

- `rc` specifies whether the method succeeded or failed. A return code of zero indicates success; a non-zero value indicates failure. If you do not supply a return code variable for the method call and the method fails, an appropriate error message will be printed to the log.
**Details**

The FIRST method returns the first data item in the hash object. If you use the `ordered: 'yes'` or `ordered: 'ascending'` argument tag in the DECLARE or _NEW_ statement when you instantiate the hash object, then the data item that is returned is the one with the 'least' key (smallest numeric value or first alphabetic character), because the data items are sorted in ascending key-value order in the hash object. Repeated calls to the NEXT method will iteratively traverse the hash object and return the data items in ascending key order. Conversely, if you use the `ordered: 'descending'` argument tag in the DECLARE or _NEW_ statement when you instantiate the hash object, then the data item that is returned is the one with the 'highest' key (largest numeric value or last alphabetic character), because the data items are sorted in descending key-value order in the hash object. Repeated calls to the NEXT method will iteratively traverse the hash object and return the data items in descending key order.

Use the LAST method to return the last data item in the hash object.

*Note:* The FIRST method sets the data variable to the value of the data item so that it is available for use after the method call.

**Example**

The following example creates a data set that contains sales data. You want to list products in order of sales. The data is loaded into a hash object and the FIRST and NEXT methods are used to retrieve the data.

```plaintext
data work.sales;
  input prod $1-6 qty $9-14;
datalines;
  banana 398487
  apple  384223
  orange 329559
;

data _null_; /* Declare hash object and read SALES data set as ordered */
  if _N_ = 1 then do;
    length prod $10;
    length qty $6;
    declare hash h(hashexp: 4, dataset: 'work.sales', ordered: 'yes');
    declare hiter iter('h');
    /* Define key and data variables */
    h.defineKey('qty');
    h.defineData('prod');
    h.defineDone();
    /* avoid uninitialized variable notes */
    call missing(qty, prod);
  end;

  /* Iterate through the hash object and output data values */
  rc = iter.first();
  do while (rc = 0);
    put prod=;
    rc = iter.next();
  end;
```
The following lines are written to the SAS log:

```sas
prod=orange
prod=banana
prod=apple
```

**See Also**

Method:

“LAST Method” on page 1650

Statements:

“DECLARE Statement” on page 1135

“_NEW_ Statement” on page 1327

“Using the Hash Iterator Object” in *SAS Language Reference: Concepts*

---

**LAST Method**

Returns the last value in the underlying hash object.

**Applies to:** Hash iterator object

**Syntax**

```sas
rc = object.LAST();
```

**Arguments**

- `rc`
  - Specifies whether the method succeeded or failed.
  - A return code of zero indicates success; a non-zero value indicates failure. If you do not supply a return code variable for the method call and the method fails, then an appropriate error message is written to the log.

- `object`
  - Specifies the name of the hash iterator object.

**Details**

The LAST method returns the last data item in the hash object. If you use the `ordered: 'yes'` or `ordered: 'ascending'` argument tag in the DECLARE or _NEW_ statement when you instantiate the hash object, then the data item that is returned is the one with the 'highest' key (largest numeric value or last alphabetic character), because the data items are sorted in ascending key-value order in the hash object. Conversely, if you use the `ordered: 'descending'` argument tag in the DECLARE or _NEW_ statement when you instantiate the hash object, then the data item that is returned is the one with the 'least' key (smallest numeric value or first alphabetic character), because the data items are sorted in descending key-value order in the hash object.
Use the FIRST method to return the first data item in the hash object.

Note: The LAST method sets the data variable to the value of the data item so that it is available for use after the method call. △

See Also

Method:
“FIRST Method” on page 1648

Statements:
“DECLARE Statement” on page 1135
“_NEW_ Statement” on page 1327
“Using the Hash Iterator Object” in *SAS Language Reference: Concepts*

---

**NEXT Method**

Returns the next value in the underlying hash object.

 Applies to: Hash iterator object

**Syntax**

```rc = object.NEXT( );```

**Arguments**

- `rc` specifies whether the method succeeded or failed.

  A return code of zero indicates success; a non-zero value indicates failure. If you do not supply a return code variable for the method call and the method fails, then an appropriate error message is written to the log.

- `object` specifies the name of the hash iterator object.

**Details**

Use the NEXT method iteratively to traverse the hash object and return the data items in key order.

The FIRST method returns the first data item in the hash object.

You can use the PREV method to return the previous data item in the hash object.

Note: The NEXT method sets the data variable to the value of the data item so that it is available for use after the method call. △

Note: If you call the NEXT method without calling the FIRST method, then the NEXT method will still start at the first item in the hash object. △
See Also

Method:

“FIRST Method” on page 1648
“PREV Method” on page 1656

Statements

“DECLARE Statement” on page 1135
“_NEW_ Statement” on page 1327
“Using the Hash Iterator Object” in SAS Language Reference: Concepts

---

**NUM_ITEMS Attribute**

Returns the number of items in the hash object.

 Applies to: Hash object

**Syntax**

```
variable_name=object.NUM_ITEMS;
```

**Arguments**

```
variable_name
```

specifies the number of items in the hash object after the method is complete.

```
object
```

specifies the name of the hash object.

**Example**

This example creates a data set and loads the data set into a hash object. An item is added to the hash object and the total number of items in the resulting hash object is returned by the NUM_ITEMS attribute.

```
data work.stock;
   input item $1-10 qty $12-14;
datalines;
broccoli 345
corn 389
potato 993
onion 730
;

data _null_
  if _N_ = 1 then do;
     length item $10;
     length qty 3;
     length totalitems 3;
```
/* Declare hash object and read STOCK data set as ordered */
declare hash myhash(hashexp: 4, dataset: "work.stock");
/* Define key and data variables */
myhash.defineKey('item');
myhash.defineData('qty');
myhash.defineDone();
end;
/* Add a key and data value to the hash object */
item = 'celery';
qty = 183;
rc = myhash.add();
if (rc ne 0) then
  put 'Add failed';
/* Use NUM_ITEMS to return updated number of items in hash object */
totalitems = myhash.num_items;
  put totalitems=;
run;

**totalitems=5** is written to the SAS log.

**OUTPUT Method**

Creates one or more data sets each of which contain the data in the hash object.

**Applies to:** Hash object

**Syntax**

```sas
rc = object OUTPUT(DATASET: 'dataset-1' <..., DATASET: 'dataset-n'>);
```

**Arguments**

- **rc**
  - specifies whether the method succeeded or failed.
  - A return code of zero indicates success; a non-zero value indicates failure. If you do not supply a return code variable for the method call and the method fails, then an appropriate error message is written to the log.

- **object**
  - specifies the name of the hash iterator object.

- **DATASET: 'dataset'**
  - specifies the name of the output data set.
  - The name of the SAS data set can be a literal or character variable. The data set name can also be enclosed in double quotation marks. Macro variables must be enclosed in double quotation marks.

**Details**

Hash object keys are not automatically stored as part of the output data set. The keys must be defined as data items by using the DEFINEDATA method to be included in the output data set.
If you use the `ordered: 'yes'` or `ordered: 'ascending'` argument tag in the DECLARE or `_NEW_` statement when you instantiate the hash object, then the data items are written to the data set in ascending key-value order. If you use the `ordered: 'descending'` argument tag in the DECLARE or `_NEW_` statement when you instantiate the hash object, then the data items are written to the data set in descending key-value order.

**Example**

Using the data set ASTRO that contains astronomical data, the following code creates a hash object with the Messier (OBJ) objects sorted in ascending order by their right-ascension (RA) values and uses the OUTPUT method to save the data to a data set.

```plaintext
data astro;
  input obj $1-4 ra $6-12 dec $14-19;
  datalines;
  M31 00 42.7 +41 16
  M71 19 53.8 +18 47
  M51 13 29.9 +47 12
  M98 12 13.8 +14 54
  M13 16 41.7 +36 28
  M39 21 32.2 +48 26
  M81 09 55.6 +69 04
  M100 12 22.9 +15 49
  M41 06 46.0 +20 44
  M44 08 40.1 +19 59
  M10 16 57.1 -04 06
  M57 18 53.6 +33 02
  M3 13 42.2 +28 23
  M22 18 36.4 -23 54
  M23 17 56.8 -19 01
  M49 12 29.8 +08 00
  M68 12 39.5 -26 45
  M17 18 20.8 -16 11
  M14 17 37.6 -03 15
  M29 20 23.9 +38 32
  M34 02 42.0 +42 47
  M82 09 55.8 +69 41
  M59 12 42.0 +11 39
  M74 01 36.7 +15 47
  M25 18 31.6 -19 15
; run;
```

```plaintext
data _null_; if _N_ = 1 then do;
  length obj $10;
  length ra $10;
  length dec $10;
  /* Read ASTRO data set as ordered */
  declare hash h(hashexp: 4, dataset:"work.astro", ordered: 'yes');
  /* Define variables RA and OBJ as key and data for hash object */
  declare hiter iter('h');
  h.defineKey('ra');
  h.defineData('ra', 'obj');
```
PROCEDURE 1:

```sas
h.defineDone();
/* avoid uninitialized variable notes */
call missing(ra, obj);
end;
/* Create output data set from hash object */
rc = h.output(dataset: 'work.out');
run;
```

```
proc print data=work.out;
  var ra obj;
  title 'Messier Objects Sorted by Right-Ascension Values';
run;
```

**Output A1.1**  
Messier Objects Sorted by Right-Ascension Values

<table>
<thead>
<tr>
<th>Obs</th>
<th>ra</th>
<th>obj</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00 42.7</td>
<td>M31</td>
</tr>
<tr>
<td>2</td>
<td>01 36.7</td>
<td>M74</td>
</tr>
<tr>
<td>3</td>
<td>02 42.0</td>
<td>M34</td>
</tr>
<tr>
<td>4</td>
<td>06 46.0</td>
<td>M41</td>
</tr>
<tr>
<td>5</td>
<td>08 40.1</td>
<td>M44</td>
</tr>
<tr>
<td>6</td>
<td>09 55.6</td>
<td>M81</td>
</tr>
<tr>
<td>7</td>
<td>09 55.8</td>
<td>M82</td>
</tr>
<tr>
<td>8</td>
<td>12 13.8</td>
<td>M98</td>
</tr>
<tr>
<td>9</td>
<td>12 22.9</td>
<td>M100</td>
</tr>
<tr>
<td>10</td>
<td>12 29.8</td>
<td>M49</td>
</tr>
<tr>
<td>11</td>
<td>12 39.5</td>
<td>M68</td>
</tr>
<tr>
<td>12</td>
<td>12 42.0</td>
<td>M59</td>
</tr>
<tr>
<td>13</td>
<td>13 29.9</td>
<td>M51</td>
</tr>
<tr>
<td>14</td>
<td>13 42.2</td>
<td>M3</td>
</tr>
<tr>
<td>15</td>
<td>16 41.7</td>
<td>M13</td>
</tr>
<tr>
<td>16</td>
<td>16 57.1</td>
<td>M10</td>
</tr>
<tr>
<td>17</td>
<td>17 37.6</td>
<td>M14</td>
</tr>
<tr>
<td>18</td>
<td>17 56.8</td>
<td>M23</td>
</tr>
<tr>
<td>19</td>
<td>18 20.8</td>
<td>M17</td>
</tr>
<tr>
<td>20</td>
<td>18 31.6</td>
<td>M25</td>
</tr>
<tr>
<td>21</td>
<td>18 36.4</td>
<td>M22</td>
</tr>
<tr>
<td>22</td>
<td>18 53.6</td>
<td>M57</td>
</tr>
<tr>
<td>23</td>
<td>19 53.8</td>
<td>M71</td>
</tr>
<tr>
<td>24</td>
<td>20 23.9</td>
<td>M29</td>
</tr>
<tr>
<td>25</td>
<td>21 32.2</td>
<td>M39</td>
</tr>
</tbody>
</table>
```

**See Also**

Method:
“DEFINEDATA Method” on page 1641

Statements:
“DECLARE Statement” on page 1135
“_NEW_ Statement” on page 1327

“Saving Hash Object Data in a Data Set” in *SAS Language Reference: Concepts*
PREV Method

Returns the previous value in the underlying hash object

Applies to: Hash iterator object

Syntax

rc=object.PREV( );

Arguments

rc specifies whether the method succeeded or failed.
A return code of zero indicates success; a non-zero value indicates failure. If you do not supply a return code variable for the method call and the method fails, then an appropriate error message is written to the log.

object specifies the name of the hash iterator object.

Details

Use the PREV method iteratively to traverse the hash object and return the data items in reverse key order.
The FIRST method returns the first data item in the hash object. The LAST method returns the last data item in the hash object.
You can use the NEXT method to return the next data item in the hash object.

Note: The PREV method sets the data variable to the value of the data item so that it is available for use after the method call.

See Also

Methods:
“FIRST Method” on page 1648
“LAST Method” on page 1650
“NEXT Method” on page 1651

Statements:
“DECLARE Statement” on page 1135
“_NEW_ Statement” on page 1327
“Using the Hash Iterator Object” in SAS Language Reference: Concepts

REMOVE Method

Removes the data that is associated with the specified key from the hash object

Applies to: Hash object
Syntax

\[ rc = \text{object}.\text{REMOVE}(\text{<KEY: keyvalue-1,..., KEY: keyvalue-n>}); \]

Arguments

- **rc** specifies whether the method succeeded or failed.
  - A return code of zero indicates success; a non-zero value indicates failure. If you do not supply a return code variable for the method call and the method fails, then an appropriate error message is written to the log.
- **object** specifies the name of the hash object.
- **KEY: keyvalue** specifies the key value whose type must match the corresponding key variable that is specified in a DEFINEKEY method call.
  - The number of “KEY: keyvalue” pairs depends on the number of key variables that you define by using the DEFINEKEY method.
- **Restriction:** If an associated hash iterator is pointing to the keyvalue, then the REMOVE method will not remove the key or data from the hash object. An error message is issued.

Details

The REMOVE method deletes both the key and the data from the hash object.

You can use the REMOVE method in one of two ways to remove the key and data in a hash object.

You can specify the key, and then use the REMOVE method as shown in the following code:

```plaintext
data _null_;  
  length k $8;  
  length d $12;  
  if _N_ = 1 then do;  
    declare hash h(hashexp: 4);  
    rc = h.defineKey('k');  
    rc = h.defineData('d');  
    rc = h.defineDone();  
    /* avoid uninitialized variable notes */  
    call missing(k, d);  
  end;  
  rc = h.add(key: 'Joyce', data: 'Ulysses');  
  /* Specify the key */  
  k = 'Joyce';  
  /* Use the REMOVE method to remove the key and data */  
  rc = h.remove();  
  if (rc = 0) then  
    put 'Key and data removed from the hash object.';  
run;
```
Alternatively, you can use a shortcut and specify the key directly in the REMOVE method call as shown in the following code:

```sas
data _null_;
  length k $8;
  length d $12;
  if _N_ = 1 then do;
    declare hash h(hashexp: 4);
    rc = h.defineKey('k');
    rc = h.defineData('d');
    rc = h.defineDone();
    /* avoid uninitialized variable notes */
    call missing(k, d);
  end;
  rc = h.add(key: 'Joyce', data: 'Ulysses');
  rc = h.add(key: 'Homer', data: 'Iliad');
  /* Specify the key in the REMOVE method parameter */
  rc = h.remove(key: 'Homer');
  if (rc = 0) then
    put 'Key and data removed from the hash object.';
run;
```

**Example**

This example illustrates how to remove a key in the hash table.

```sas
/* Generate test data */
data x;
  do k = 65 to 70;
    d = byte (k);
    output;
  end;
run;

data _null_;
  length k 8 d $1;
  /*define the hash table and iterator*/
  declare hash H (dataset:'x', ordered:'a');
  H.defineKey ('k');
  H.defineData ('k', 'd');
  H.defineDone ()
  call missing (k,d);
  declare hiter HI ('H');
  /* Use this logic to remove a key in the hash table
  when an iterator is pointing to that key*/
  do while (hi.next() = 0);
    if flag then rc=h.remove(key:key);
    if d = 'C' then do;
      key=k;
      flag=1;
    end;
  end;
```
DATA Step Object Attributes and Methods  △  REPLACE Method  1659

/*****************************/
/* Note that the following DO WHILE loop will */
/* result in the error: */
/* */
/* ERROR: Read Access Violation In Task [DATASTEP]*/ *
/* */
/* because it causes the key to which the iterator */
/* is pointing to be removed. */
/* */
/* do while (hi.next() = 0); */
/* if d = 'C' then rc_remove = h.remove(); */
/* end; */
/*****************************/

rc = h.output(dataset: 'work.out');
stop;
run;

proc print;
run;

The following output shows that the key and data for the third object (key=67, data=C) is deleted.

Output A1.2

<table>
<thead>
<tr>
<th>The SAS System</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obs</td>
<td>k</td>
</tr>
<tr>
<td>1</td>
<td>65</td>
</tr>
<tr>
<td>2</td>
<td>66</td>
</tr>
<tr>
<td>3</td>
<td>68</td>
</tr>
<tr>
<td>4</td>
<td>69</td>
</tr>
<tr>
<td>5</td>
<td>70</td>
</tr>
</tbody>
</table>

See Also

Methods:
“ADD Method” on page 1637
“DEFINEKEY Method” on page 1644
“REPLACE Method” on page 1659
“Replacing and Removing Data” in SAS Language Reference: Concepts

REPLACE Method

Replaces the data that is associated with the specified key with new data
Applies to: Hash object

Syntax

\[ rc = \text{object}.\text{REPLACE}(<\text{KEY: keyvalue-1,\ldots},\text{KEY: keyvalue-n, DATA: datavalue-1,\ldots}, \text{DATA: datavalue-n}>) ; \]

Arguments

- \( rc \) specifies whether the method succeeded or failed. A return code of zero indicates success; a non-zero value indicates failure. If you do not supply a return code variable for the method call and the method fails, then an appropriate error message is written to the log.

- \( \text{object} \) specifies the name of the hash object.

- \( \text{KEY: keyvalue} \) specifies the key value whose type must match the corresponding key variable that is specified in a DEFINEKEY method call. The number of “\( \text{KEY: keyvalue} \)” pairs depends on the number of key variables that you define by using the DEFINEKEY method.

- \( \text{DATA: datavalue} \) specifies the data value whose type must match the corresponding data variable that is specified in a DEFINEDATA method call. The number of “\( \text{DATA: datavalue} \)” pairs depends on the number of data variables that you define by using the DEFINEDATA method.

Details

You can use the REPLACE method in one of two ways to replace data in a hash object. You can define the key and data item, and then use the REPLACE method as shown in the following code. In this example the data for the key 'Rottwlr' is changed from '1st' to '2nd'.

```plaintext
data work.show;
   input brd $1-10 plc $12-14;
datalines;
  Terrier   2nd
  LabRetr   3rd
  Rottwlr   1st
  Collie    bis
  ChinsCrstd 2nd
  Newfn1nd  3rd
;

data _null_
   length brd $12;
   length plc $8;
   if _N_ = 1 then do;
     declare hash h(hashexp: 4, dataset: 'work.show');
     rc = h.defineKey('brd');
```

```plaintext
```
rc = h.defineData(‘plc’);
rc = h.defineDone();
end;

/* Specify the key and new data value */
brd = ‘Rottwlr’;
plc = ‘2nd’;
/* Call the REPLACE method to replace the data value */
rc = h.replace();
run;

Alternatively, you can use a shortcut and specify the key and data directly in the REPLACE method call as shown in the following code:

data work.show;
  input brd $1-10 plc $12-14;
datalines;
  Terrier 2nd
  LabRetr 3rd
  Rottwlr 1st
  Collie bis
  ChinsCrstd 2nd
  Newfn1nd 3rd
;

data _null_;  
  length brd $12;
  length plc $8;
  if _N_ = 1 then do;
    declare hash h(hashexp: 4, dataset: ‘work.show’);
    rc = h.defineKey(‘brd’);
    rc = h.defineData(‘plc’);
    rc = h.defineDone();
    /* avoid uninitialized variable notes */
    call missing(brd, plc);
  end;

  /* Specify the key and new data value in the REPLACE method */
  rc = h.replace(key: ‘Rottwlr’, data: ‘2nd’);
run;

Note: If you call the REPLACE method and the key is not found, then the key and data are added to the hash object. △

See Also

Methods:

“DEFINEDATA Method” on page 1641
“DEFINEKEY Method” on page 1644
“REMOVE Method” on page 1656
“Replacing and Removing Data” in SAS Language Reference: Concepts
APPENDIX 2

DATA Step Debugger

Introduction 1664
  Definition: What is Debugging? 1664
  Definition: The DATA Step Debugger 1664
Basic Usage 1665
  How a Debugger Session Works 1665
  Using the Windows 1665
  Entering Commands 1665
  Working with Expressions 1666
  Assigning Commands to Function Keys 1666
Advanced Usage: Using the Macro Facility with the Debugger 1666
  Using Macros as Debugging Tools 1666
  Creating Customized Debugging Commands with Macros 1666
  Debugging a DATA Step Generated by a Macro 1667
Examples 1667
  Example 1: Debugging a Simple DATA Step 1667
    Discovering a Problem 1667
    Using the DEBUG Option 1668
    Examining Data Values after the First Iteration 1669
    Examining Data Values after the Second Iteration 1670
    Ending the Debugger 1671
    Correcting the DATA Step 1671
  Example 2: Working with Formats 1672
  Example 3: Debugging DO Loops 1677
  Example 4: Examining Formatted Values of Variables 1677
Commands 1678
  List of Debugger Commands 1678
  Debugger Commands by Category 1678
Dictionary 1679
  BREAK 1679
  CALCULATE 1682
  DELETE 1682
  DESCRIBE 1683
  ENTER 1684
  EXAMINE 1685
  GO 1686
  HELP 1687
  JUMP 1687
  LIST 1688
  QUIT 1689
  SET 1690
  STEP 1691
**Definition: What is Debugging?**

Debugging is the process of removing logic errors from a program. Unlike syntax errors, logic errors do not stop a program from running. Instead, they cause the program to produce unexpected results. For example, if you create a DATA step that keeps track of inventory, and your program shows that you are out of stock but your warehouse is full, you have a logic error in your program.

To debug a DATA step, you could

- copy a few lines of the step into another DATA step, execute it, and print the results of those statements
- insert PUT statements at selected places in the DATA step, submit the step, and examine the values that are displayed in the SAS log.
- use the DATA step debugger.

While the SAS log can help you identify data errors, the DATA step debugger offers you an easier, interactive way to identify logic errors, and sometimes data errors, in DATA steps.

**Definition: The DATA Step Debugger**

The DATA step debugger is part of Base SAS software and consists of windows and a group of commands. By issuing commands, you can execute DATA step statements one by one and pause to display the resulting variable values in a window. By observing the results that are displayed, you can determine where the logic error lies. Because the debugger is interactive, you can repeat the process of issuing commands and observing the results as many times as needed in a single debugging session. To invoke the debugger, add the DEBUG option to the DATA statement and execute the program.

The DATA step debugger enables you to perform the following tasks:

- execute statements one by one or in groups
- bypass execution of one or more statements
- suspend execution at selected statements, either in each iteration of DATA step statements or on a condition you specify, and resume execution on command
- monitor the values of selected variables and suspend execution at the point a value changes
- display the values of variables and assign new values to them
- display the attributes of variables
- receive help for individual debugger commands
- assign debugger commands to function keys
- use the macro facility to generate customized debugger commands.
Basic Usage

How a Debugger Session Works

When you submit a DATA step with the DEBUG option, SAS compiles the step, displays the debugger windows, and pauses until you enter a debugger command to begin execution. If you begin execution with the GO command, for example, SAS executes each statement in the DATA step. To suspend execution at a particular line in the DATA step, use the BREAK command to set breakpoints at statements you select. Then issue the GO command. The GO command starts or resumes execution until the breakpoint is reached.

To execute the DATA step one statement at a time or a few statements at a time, use the STEP command. By default, the STEP command is mapped to the ENTER key.

In a debugging session, statements in a DATA step can iterate as many times as they would outside the debugging session. When the last iteration has finished, a message appears in the DEBUGGER LOG window.

You cannot restart DATA step execution in a debugging session after the DATA step finishes executing. You must resubmit the DATA step in your SAS session. However, you can examine the final values of variables after execution has ended.

You can debug only one DATA step at a time. You can use the debugger only with a DATA step, and not with a PROC step.

Using the Windows

The DATA step debugger contains two primary windows, the DEBUGGER LOG and the DEBUGGER SOURCE windows. The windows appear when you execute a DATA step with the DEBUG option.

The DEBUGGER LOG window records the debugger commands you issue and their results. The last line is the debugger command line, where you issue debugger commands. The debugger command line is marked with a greater than (>) prompt.

The DEBUGGER SOURCE window contains the SAS statements that comprise the DATA step you are debugging. The window enables you to view your position in the DATA step as you debug your program. In the window, the SAS statements have the same line numbers as they do in the SAS log.

You can enter windowing environment commands on the window command lines. You can also execute commands by using function keys.

Entering Commands

Enter DATA step debugger commands on the debugger command line. For a list of commands and their descriptions, refer to “Debugger Commands by Category” on page 1678. Follow these rules when you enter a command:

- A command can occupy only one line (except for a DO group).
- A DO group can extend over more than one line.
- To enter multiple commands, separate the commands with semicolons:

```plaintext
examine _all_; set letter='bill'; examine letter
```
Working with Expressions

All SAS operators that are described in “SAS Operators in Expressions” in SAS Language Reference: Concepts, are valid in debugger expressions. Debugger expressions cannot contain functions.

A debugger expression must fit on one line. You cannot continue an expression on another line.

Assigning Commands to Function Keys

To assign debugger commands to function keys, open the Keys window. Position your cursor in the Definitions column of the function key you want to assign, and begin the command with the term DSD. To assign more than one command to a function key, enclose the commands (separated by semicolons) in quotation marks. Be sure to save your changes. These examples show commands assigned to function keys:

- dsd step3
- dsd ‘examine cost saleprice; go 120;’

Advanced Usage: Using the Macro Facility with the Debugger

You can use the SAS macro facility with the debugger to invoke macros from the DEBUGGER LOG command line. You can also define macros and use macro program statements, such as %LET, on the debugger command line.

Using Macros as Debugging Tools

Macros are useful for storing a series of debugger commands. Executing the macro at the DEBUGGER LOG command line then generates the entire series of debugger commands. You can also use macros with parameters to build different series of debugger commands based on various conditions.

Creating Customized Debugging Commands with Macros

You can create a customized debugging command by defining a macro on the DEBUGGER LOG command line. Then invoke the macro from the command line. For example, to examine the variable COST, to execute five statements, and then to examine the variable DURATION, define the following macro (in this case the macro is called EC). Note that the example uses the alias for the EXAMINE command.

```sas
%macro ec; ex cost; step 5; ex duration; %mend ec;
```

To issue the commands, invoke macro EC from the DEBUGGER LOG command line:

```sas
%ec
```

The DEBUGGER LOG displays the value of COST, executes the next five statements, and then displays the value of DURATION.
Note: Defining a macro on the DEBUGGER LOG command line allows you to use the macro only during the current debugging session, because the macro is not permanently stored. To create a permanently stored macro, use the Program Editor.

Debugging a DATA Step Generated by a Macro

You can use a macro to generate a DATA step, but debugging a DATA step that is generated by a macro can be difficult. The SAS log displays a copy of the macro, but not the DATA step that the macro generated. If you use the DEBUG option at this point, the text that the macro generates appears as a continuous stream to the debugger. As a result, there are no line breaks where execution can pause.

To debug a DATA step that is generated by a macro, use the following steps:

1. Use the MPRINT and MFILE system options when you execute your program.
2. Assign the fileref MPRINT to an existing external file. MFILE routes the program output to the external file. Note that if you rerun your program, current output appends to the previous output in your file.
3. Invoke the macro from a SAS session.
4. In the Program Editor window, issue the INCLUDE command or use the File menu to open your external file.
5. Add the DEBUG option to the DATA statement and begin a debugging session.
6. When you locate the logic error, correct the portion of the macro that generated that statement or statements.

Examples

Example 1: Debugging a Simple DATA Step

This example shows how to debug a DATA step when output is missing.

Discovering a Problem

This program creates information about a travel tour group. The data files contain two types of records. One type contains the tour code, and the other type contains customer information. The program creates a report listing tour number, name, age, and sex for each customer.

```/* first execution */
data tours (drop=type);
  input @1 type $ @;
  if type='H' then do;
    input @3 Tour $20.;
    return;
  end;
  else if type='P' then do;
    input @3 Name $10. Age 2. +1 Sex $1.;
    output;
  end;
```
datalines;
H Tour 101
P Mary E 21 F
P George S 45 M
P Susan K 3 F
H Tour 102
P Adelle S 79 M
P Walter P 55 M
P Fran I 63 F
;
proc print data=tours;
title 'Tour List';
run;

<table>
<thead>
<tr>
<th>Obs</th>
<th>Tour</th>
<th>Name</th>
<th>Age</th>
<th>Sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mary E</td>
<td>21</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>George S</td>
<td>45</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Susan K</td>
<td>3</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Adelle S</td>
<td>79</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Walter P</td>
<td>55</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Fran I</td>
<td>63</td>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>

The program executes without error, but the output is unexpected. The output does not contain values for the variable Tour. Viewing the SAS log will not help you debug the program because the data are valid and no errors appear in the log. To help identify the logic error, run the DATA step again using the DATA step debugger.

Using the DEBUG Option

To invoke the DATA step debugger, add the DEBUG option to the DATA statement and resubmit the DATA step:

data tours (drop=type) / debug;

The following display shows the resulting two debugger windows.
DATA Step Debugger  Example 1: Debugging a Simple DATA Step 1669

The upper window is the DEBUGGER LOG window. Issue debugger commands in this window by typing commands on the debugger command line (the bottom line, marked by a >). The debugger displays the command and results in the upper part of the window.

The lower window is the DEBUGGER SOURCE window. It displays the DATA step submitted with the DEBUG option. Each line in the DATA step is numbered with the same line number used in the SAS log. One line appears in reverse video (or other highlighting, depending on your monitor). DATA step execution pauses just before the execution of the highlighted statement.

At the beginning of your debugging session, the first executable line after the DATA statement is highlighted. This means that SAS has compiled the step and will begin to execute the step at the top of the DATA step loop.

Examining Data Values after the First Iteration

To debug a DATA step, create a hypothesis about the logic error and test it by examining the values of variables at various points in the program. For example, issue the EXAMINE command from the debugger command line to display the values of all variables in the program data vector before execution begins:

```
examine _all_
```

Note: Most debugger commands have abbreviations, and you can assign commands to function keys. The examples in this section, however, show the full command name to help you find the commands in “Debugger Commands by Category” on page 1678.

When you press ENTER, the following display appears:

The values of all variables appear in the DEBUGGER LOG window. SAS has compiled, but not yet executed, the INPUT statement.

Use the STEP command to execute the DATA step statements one at a time. By default, the STEP command is assigned to the ENTER key. Press ENTER repeatedly to step through the first iteration of the DATA step, and stop when the RETURN statement in the program is highlighted in the DEBUGGER SOURCE window.
Because Tour information was missing in the program output, enter the EXAMINE command to view the value of the variable Tour for the first iteration of the DATA step.

```sas
examine tour
```

The following display shows the results:

![SAS Debugger Log]

The variable Tour contains the value Tour 101, showing you that Tour was read. The first iteration of the DATA step worked as intended. Press ENTER to reach the top of the DATA step.

### Examining Data Values after the Second Iteration

You can use the BREAK command (also known as setting a breakpoint) to suspend DATA step execution at a particular line you designate. In this example, suspend execution before executing the ELSE statement by setting a breakpoint at line 7.

```sas
break 7
```

When you press ENTER, an exclamation point appears at line 7 in the DEBUGGER SOURCE window to mark the breakpoint:

![SAS Debugger Source]

Execute the GO command to continue DATA step execution until it reaches the breakpoint (in this case, line 7):

```sas
go
```

The following display shows the result:
SAS suspended execution just before the ELSE statement in line 7. Examine the values of all the variables to see their status at this point.

```
examine _all_
```

The following display shows the values:

You expect to see a value for Tour, but it does not appear. The program data vector gets reset to missing values at the beginning of each iteration and therefore does not retain the value of Tour. To solve the logic problem, you need to include a RETAIN statement in the SAS program.

**Ending the Debugger**

To end the debugging session, issue the QUIT command on the debugger command line:

```
quit
```

The debugging windows disappear, and the original SAS session resumes.

**Correcting the DATA Step**

Correct the original program by adding the RETAIN statement. Delete the DEBUG option from the DATA step, and resubmit the program:

```
/* corrected version */
data tours (drop=type);
```
Example 2: Working with Formats

This example shows how to debug a program when you use format statements to format dates. The following program creates a report that lists travel tour dates for specific countries.

```plaintext
options yearcutoff=1920;

data tours;
  length Country $ 10;
  Duration=end-start;
datalines;
Italy  033000  041300
Brazil  021900  022800
```

The values for Tour now appear in the output:

```
Obs  Tour Name Age Sex
1    Tour 101 Mary E 21 F
2    Tour 101 George S 45 M
3    Tour 101 Susan K  3 F
4    Tour 102 Adelle S 79 M
5    Tour 102 Walter P 55 M
6    Tour 102 Fran I  63 F
```
DATA Step Debugger  △  Example 2: Working with Formats  1673

Japan 052200 061500
Venezuela 110300 11800
Australia 122100 011501
;

proc print data=tours;
   format start end date9.;
   title 'Tour Duration';
run;

<table>
<thead>
<tr>
<th>Obs</th>
<th>Country</th>
<th>Start</th>
<th>End</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Italy</td>
<td>30MAR2000</td>
<td>13APR2000</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>Brazil</td>
<td>19FEB2000</td>
<td>28FEB2000</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>Japan</td>
<td>22MAY2000</td>
<td>15JUN2000</td>
<td>24</td>
</tr>
<tr>
<td>4</td>
<td>Venezuela</td>
<td>03NOV2000</td>
<td>18JAN2000</td>
<td>-290</td>
</tr>
<tr>
<td>5</td>
<td>Australia</td>
<td>21DEC2000</td>
<td>15JAN2001</td>
<td>25</td>
</tr>
</tbody>
</table>

The value of Duration for the tour to Venezuela shows a negative number, -290 days. To help identify the error, run the DATA step again using the DATA step debugger. SAS displays the following debugger windows:

At the DEBUGGER LOG command line, issue the EXAMINE command to display the values of all variables in the program data vector before execution begins:

```plaintext
examine _all_
```

Initial values of all variables appear in the DEBUGGER LOG window. SAS has not yet executed the INPUT statement.

Press ENTER to issue the STEP command. SAS executes the INPUT statement, and the assignment statement is now highlighted.

Issue the EXAMINE command to display the current value of all variables:

```plaintext
examine _all_
```
The following display shows the results:

Because a problem exists with the Venezuela tour, suspend execution before the assignment statement when the value of Country equals Venezuela. Set a breakpoint to do this:

```
break 4 when country='Venezuela'
```

Execute the GO command to resume program execution:

```
go
```

SAS stops execution when the country name is Venezuela. You can examine Start and End tour dates for the Venezuela trip. Because the assignment statement is highlighted (indicating that SAS has not yet executed that statement), there will be no value for Duration.

Execute the EXAMINE command to view the value of the variables after execution:

```
examine _all_
```
The following display shows the results:

To view formatted SAS dates, issue the EXAMINE command using the DATEw. format:

```
examine start date7. end date7.
```

The following display shows the results:

Because the tour ends on November 18, 2000, and not on January 18, 2000, there is an error in the variable End. Examine the source data in the program and notice that the value for End has a typographical error. By using the SET command, you can temporarily set the value of End to November 18 to see if you get the anticipated result. Issue the SET command using the DDMMYYw. format:

```
set end='18nov00'd
```
Press ENTER to issue the STEP command and execute the assignment statement. Issue the EXAMINE command to view the tour date and Duration fields:

```
examine start date7. end date7. duration
```

The following display shows the results:

```
The Start, End, and Duration fields contain correct data.
End the debugging session by issuing the QUIT command on the DEBUGGER LOG command line. Correct the original data in the SAS program, delete the DEBUG option, and resubmit the program.

```/* corrected version */
options yearcutoff=1920;

data tours;
    length Country $ 10;
        duration=end-start;
datalines;
Italy 033000 041300
Brazil 021900 022800
Japan 052200 061500
Venezuela 110300 111800
Australia 122100 011501
;

proc print data=tours;
    format start end date9.;
    title 'Tour Duration';
run;
```
**Example 3: Debugging DO Loops**

An iterative DO, DO WHILE, or DO UNTIL statement can iterate many times during a single iteration of the DATA step. When you debug DO loops, you can examine several iterations of the loop by using the AFTER option in the BREAK command. The AFTER option requires a number that indicates how many times the loop will iterate before it reaches the breakpoint. The BREAK command then suspends program execution. For example, consider this data set:

```sas
data new / debug;
  set old;
  do i=1 to 20;
    newtest=oldtest+i;
    output;
  end;
run;
```

To set a breakpoint at the assignment statement (line 4 in this example) after every 5 iterations of the DO loop, issue this command:

```
break 4 after 5
```

When you issue the GO commands, the debugger suspends execution when I has the values of 5, 10, 15, and 20 in the DO loop iteration.

In an iterative DO loop, select a value for the AFTER option that can be divided evenly into the number of iterations of the loop. For example, in this DATA step, 5 can be evenly divided into 20. When the DO loop iterates the second time, I again has the values of 5, 10, 15, and 20.

If you do not select a value that can be evenly divided (such as 3 in this example), the AFTER option causes the debugger to suspend execution when I has the values of 3, 6, 9, 12, 15, and 18. When the DO loop iterates the second time, I has the values of 1, 4, 7, 10, 13, and 16.

**Example 4: Examining Formatted Values of Variables**

You can use a SAS format or a user-created format when you display a value with the EXAMINE command. For example, assume the variable BEGIN contains a SAS date value. To display the day of the week and date, use the SAS WEEKDATEw format with EXAMINE:

```
examine begin weekdate17.
```
When the value of BEGIN is 033001, the debugger displays

Sun, Mar 30, 2001

As another example, you can create a format named SIZE:

```plaintext
proc format;
  value size 1-5='small'
    6-10='medium'
    11-high='large';
run;
```

To debug a DATA step that applies the format SIZE to the variable STOCKNUM, use the format with EXAMINE:

```plaintext
examine stocknum size.
```

When the value of STOCKNUM is 7, for example, the debugger displays

STOCKNUM = medium

---

## Commands

### List of Debugger Commands

- BREAK
- CALCULATE
- DELETE
- DESCRIBE
- ENTER
- EXAMINE
- GO
- HELP

JUMP
LIST
QUIT
SET
STEP
SWAP
TRACE
WATCH

### Debugger Commands by Category

<table>
<thead>
<tr>
<th>Table A2.1 Categories and Descriptions of Debugger Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Category</strong></td>
</tr>
<tr>
<td>Controlling Program Execution</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
Controlling the Windows

- "STEP" on page 1691: Executes statements one at a time in the active program
- "HELP" on page 1687: Displays information about debugger commands
- "SWAP" on page 1692: Switches control between the SOURCE window and the LOG window

Manipulating DATA Step Variables

- "CALCULATE" on page 1682: Evaluates a debugger expression and displays the result
- "DESCRIBE" on page 1683: Displays the attributes of one or more variables
- "EXAMINE" on page 1685: Displays the value of one or more variables
- "SET" on page 1690: Assigns a new value to a specified variable

Manipulating Debugging Requests

- "BREAK" on page 1679: Suspends program execution at an executable statement
- "DELETE" on page 1682: Deletes breakpoints or the watch status of variables in the DATA step
- "LIST" on page 1688: Displays all occurrences of the item that is listed in the argument
- "TRACE" on page 1692: Controls whether the debugger displays a continuous record of the DATA step execution
- "WATCH" on page 1693: Suspends execution when the value of a specified variable changes

Tailoring the Debugger

- "ENTER" on page 1684: Assigns one or more debugger commands to the ENTER key

Terminating the Debugger

- "QUIT" on page 1689: Terminates a debugger session

---

**Dictionary**

**BREAK**

Suspends program execution at an executable statement

**Category:** Manipulating Debugging Requests

**Alias:** B

**Syntax**

```
BREAK location <AFTER count> <WHEN expression> <DO group >
```
Arguments

**location**
- Specifies where to set a breakpoint. Location must be one of these:
  - **label**
    - A statement label. The breakpoint is set at the statement that follows the label.
  - **line-number**
    - The number of a program line at which to set a breakpoint.
  - *** the current line.

**AFTER count**
- Honors the breakpoint each time the statement has been executed count times. The counting is continuous. That is, when the AFTER option applies to a statement inside a DO loop, the count continues from one iteration of the loop to the next. The debugger does not reset the count value to 1 at the beginning of each iteration.

  If a BREAK command contains both AFTER and WHEN, AFTER is evaluated first. If the AFTER count is satisfied, the WHEN expression is evaluated.

  **Tip:** The AFTER option is useful in debugging DO loops.

**WHEN expression**
- Honors a breakpoint when the expression is true.

**DO group**
- Is one or more debugger commands enclosed by a DO and an END statement. The syntax of the DO group is

  DO; command-1 < ... ; command-n; >END;

  **command**
  - Specifies a debugger command. Separate multiple commands by semicolons.

  A DO group can span more than one line and can contain IF-THEN/ELSE statements, as shown:

  **IF** expression **THEN** command; **<ELSE command;>**
  **IF** expression **THEN** DO group; **<ELSE DO group;>**

  If evaluates an expression. When the condition is true, the debugger command or DO group in the THEN clause executes. An optional ELSE command gives an alternative action if the condition is not true. You can use these arguments with IF:

  **expression**
  - Specifies a debugger expression. A non-zero, nonmissing result causes the expression to be true. A result of zero or missing causes the expression to be false.

  **command**
  - Specifies a single debugger command.

  **DO group**
  - Specifies a DO group.

Details

The BREAK command suspends execution of the DATA step at a specified statement. Executing the BREAK command is called *setting a breakpoint.*
When the debugger detects a breakpoint, it

- checks the AFTER count value, if present, and suspends execution if count breakpoint activations have been reached
- evaluates the WHEN expression, if present, and suspends execution if the condition that is evaluated is true
- suspends execution if neither an AFTER nor a WHEN clause is present
- displays the line number at which execution is suspended
- executes any commands that are present in a DO group
- returns control to the user with a > prompt.

If a breakpoint is set at a source line that contains more than one statement, the breakpoint applies to each statement on the source line. If a breakpoint is set at a line that contains a macro invocation, the debugger breaks at each statement generated by the macro.

**Examples**

- Set a breakpoint at line 5 in the current program:
  
  ```
  b 5
  ```

- Set a breakpoint at the statement after the statement label\n  `eoflabel`:
  
  ```
  eoflabel:
  b eoflabel
  ```

- Set a breakpoint at line 45 that will be honored after every third execution of line 45:
  
  ```
  b 45 after 3
  ```

- Set a breakpoint at line 45 that will be honored after every third execution of that line only when the values of both DIVISOR and DIVIDEND are 0:
  
  ```
  b 45 after 3
  when (divisor=0 and dividend=0)
  ```

- Set a breakpoint at line 45 of the program and examine the values of variables NAME and AGE:
  
  ```
  b 45 do; ex name age; end;
  ```

- Set a breakpoint at line 15 of the program. If the value of DIVISOR is greater than 3, execute STEP; otherwise, display the value of DIVIDEND.
  
  ```
  b 15 do; if divisor>3 then st;
  else ex dividend; end;
  ```

**See Also**

Commands:

- “DELETE” on page 1682
- “WATCH” on page 1693
CALCULATE

Evaluates a debugger expression and displays the result

Category: Manipulating DATA Step Variables

Syntax

CALC expression

Arguments

expression specifies any debugger expression.

Restriction: Debugger expressions cannot contain functions.

Details

The CALCULATE command evaluates debugger expressions and displays the result. The result must be numeric.

Examples

- Add 1.1, 1.2, 3.4 and multiply the result by 0.5:
  \[
  \text{calc (1.1+1.2+3.4)*0.5}
  \]
- Calculate the sum of STARTAGE and DURATION:
  \[
  \text{calc startage+duration}
  \]
- Calculate the values of the variable SALE minus the variable DOWNPAY and then multiply the result by the value of the variable RATE. Divide that value by 12 and add 50:
  \[
  \text{calc (((sale-downpay)*rate)/12)+50}
  \]

See Also

“Working with Expressions” on page 1666 for information on debugger expressions

DELETE

Deletes breakpoints or the watch status of variables in the DATA step

Category: Manipulating Debugging Requests

Alias: D
Data Step Debugger

Syntax

DELETE BREAK location
DELETE WATCH variable(s) | _ALL_

Arguments

BREAK
deletes breakpoints.
Alias: B

location specifies a breakpoint location to be deleted. Location can have one of these values:

_ALL_ all current breakpoints in the DATA step.
label the statement after a statement label.
line-number the number of a program line.
* the breakpoint from the current line.

WATCH
deletes watched status of variables.
Alias: W

variable names one or more watched variables for which the watch status is deleted.
_ALL_ specifies that the watch status is deleted for all watched variables.

Examples

- Delete the breakpoint at the statement label
  
  `d b eoflabel`

- Delete the watch status from the variable ABC in the current DATA step:
  
  `d w abc`

See Also

Commands:
“BREAK” on page 1679
“WATCH” on page 1693
Category: Manipulating DATA Step Variables
Alias: DESC

Syntax

DESCRIBE variable(s) | _ALL_

Arguments

variable
identifies a DATA step variable.

_ALL_
indicates all variables that are defined in the DATA step.

Details

The DESCRIBE command displays the attributes of one or more specified variables. DESCRIBE reports the name, type, and length of the variable, and, if present, the informat, format, or variable label.

Examples

- Display the attributes of variable ADDRESS:
  desc address

- Display the attributes of array element ARR[i + j]:
  desc arr[i+j]

ENTER

Assigns one or more debugger commands to the ENTER key

Category: Tailoring the Debugger

Syntax

ENTER <command-1 <... ; command-n>>

Arguments

command
specifies a debugger command.

Default: STEP 1
Details
The ENTER command assigns one or more debugger commands to the ENTER key. Assigning a new command to the ENTER key replaces the existing command assignment. If you assign more than one command, separate the commands with semicolons.

Examples

- Assign the command STEP 5 to the ENTER key:
  
  `enter st 5`

- Assign the commands EXAMINE and DESCRIBE, both for the variable CITY, to the ENTER key:
  
  `enter ex city; desc city`

EXAMINE

Displays the value of one or more variables

Category: Manipulating DATA Step Variables

Alias: E

Syntax

EXAMINE variable-1 <format-1> <...variable-n <format-n>>

EXAMINE _ALL_ <format>

Arguments

- **variable**
  - identifies a DATA step variable.

- **format**
  - identifies a SAS format or a user-created format.

- **_ALL_**
  - identifies all variables that are defined in the current DATA step.

Details

The EXAMINE command displays the value of one or more specified variables. The debugger displays the value using the format currently associated with the variable, unless you specify a different format.
Examples

- Display the values of variables N and STR:
  
  ```ex n str```

- Display the element \(i\) of the array TESTARR:
  
  ```ex testarr(i)```

- Display the elements \(i+1, j*2, k-3\) of the array CRR:
  
  ```ex crr(i+1); ex crr(j*2); ex crr(k-3)```

- Display the SAS date variable T_DATE with the DATE7. format:
  
  ```ex t_date date7.```

- Display the values of all elements in array NEWARR:
  
  ```ex newarr{*}```

See Also

Command:

“DESCRIBE” on page 1683

GO

**Starts or resumes execution of the DATA step**

**Category:** Controlling Program Execution  
**Alias:** G

---

**Syntax**

```
GO <line-number | label>
```

**Without Arguments**

If you omit arguments, GO resumes execution of the DATA step and executes its statements continuously until a breakpoint is encountered, until the value of a watched variable changes, or until the DATA step completes execution.

**Arguments**

- **line-number**
  
gives the number of a program line at which execution is to be suspended next.

- **label**
  
is a statement label. Execution is suspended at the statement following the statement label.
Details
The GO command starts or resumes execution of the DATA step. Execution continues until all observations have been read, a breakpoint specified in the GO command is reached, or a breakpoint set earlier with a BREAK command is reached.

Examples

- Resume executing the program and execute its statements continuously:
  `g`
- Resume program execution and then suspend execution at the statement in line 104:
  `g 104`

See Also

Commands:
- “JUMP” on page 1687
- “STEP” on page 1691
Syntax

JUMP line-number | label

Arguments

line-number
indicates the number of a program line at which to restart the suspended program.

label
is a statement label. Execution resumes at the statement following the label.

Details

The JUMP command moves program execution to the specified location without executing intervening statements. After executing JUMP, you must restart execution with GO or STEP. You can jump to any executable statement in the DATA step.

CAUTION:
Do not use the JUMP command to jump to a statement inside a DO loop or to a label that is the target of a LINK-RETURN group. In such cases you bypass the controls set up at the beginning of the loop or in the LINK statement, and unexpected results can appear.

JUMP is useful in two situations:

□ when you want to bypass a section of code that is causing problems in order to concentrate on another section. In this case, use the JUMP command to move to a point in the DATA step after the problematic section.

□ when you want to re-execute a series of statements that have caused problems. In this case, use JUMP to move to a point in the DATA step before the problematic statements and use the SET command to reset values of the relevant variables to the values they had at that point. Then re-execute those statements with STEP or GO.

Examples

□ Jump to line 5: j 5

See Also

Commands:
“GO” on page 1686
“STEP” on page 1691

LIST

Displays all occurrences of the item that is listed in the argument

Category: Manipulating Debugging Requests
Alias:  L

Syntax
LIST _ALL_ | BREAK | DATASETS | FILES | INFILES | WATCH

Arguments

_ALL_
  displays the values of all items.

BREAK
  displays breakpoints.
  Alias:  B

DATASETS
  displays all SAS data sets used by the current DATA step.

FILES
  displays all external files to which the current DATA step writes.

INFILES
  displays all external files from which the current DATA step reads.

WATCH
  displays watched variables.
  Alias:  W

Examples

  □ List all breakpoints, SAS data sets, external files, and watched variables for the current DATA step:

  l _all_

  □ List all breakpoints in the current DATA step:

  l b

See Also

Commands:
  “BREAK” on page 1679
  “DELETE” on page 1682
  “WATCH” on page 1693
Category: Terminating the Debugger
Alias: Q

Syntax
QUIT

Without Arguments
The QUIT command terminates a debugger session and returns control to the SAS session.

Details
SAS creates data sets built by the DATA step that you are debugging. However, when you use QUIT to exit the debugger, SAS does not add the current observation to the data set.

You can use the QUIT command at any time during a debugger session. After you end the debugger session, you must resubmit the DATA step with the DEBUG option to begin a new debugging session; you cannot resume a session after you have ended it.

SET

Assigns a new value to a specified variable
Category: Manipulating DATA Step Variables
Alias: None

Syntax
SET variable=expression

Arguments

variable
specifies the name of a DATA step variable or an array reference.

eexpression
is any debugger expression.

Tip: Expression can contain the variable name that is used on the left side of the equal sign. When a variable appears on both sides of the equal sign, the debugger uses the original value on the right side to evaluate the expression and stores the result in the variable on the left.

Details
The SET command assigns a value to a specified variable. When you detect an error during program execution, you can use this command to assign new values to variables. This enables you to continue the debugging session.
Examples

- Set the variable A to the value of 3:
  
  ```
  set a=3
  ```

- Assign to the variable B the value \texttt{12345} concatenated with the previous value of B:
  
  ```
  set b='12345' || b
  ```

- Set array element ARR[1] to the result of the expression \texttt{a+3}:
  
  ```
  set arr{1}=a+3
  ```

- Set array element CRR[1,2,3] to the result of the expression \texttt{crr[1,1,2] + crr[1,1,3]}:
  
  ```
  set crr{1,2,3} = crr{1,1,2} + crr{1,1,3}
  ```

- Set variable A to the result of the expression \texttt{a+c*3}:
  
  ```
  set a=a+c*3
  ```

---

**STEP**

Executes statements one at a time in the active program

Category: Controlling Program Execution

Alias: ST

---

**Syntax**

```
STEP <n>
```

**Without Arguments**

STEP executes one statement.

**Arguments**

- \texttt{n}

  specifies the number of statements to execute.

**Details**

The \texttt{STEP} command executes statements in the DATA step, starting with the statement at which execution was suspended. When you issue a \texttt{STEP} command, the debugger:

- executes the number of statements that you specify
- displays the line number
- returns control to the user and displays the > prompt.

\textit{Note:} By default, you can execute the \texttt{STEP} command by pressing the ENTER key.
See Also

Commands:
“GO” on page 1686
“JUMP” on page 1687

SWAP

Switches control between the SOURCE window and the LOG window

Category: Controlling the Windows
Alias: None

Syntax
SWAP

Without Arguments
The SWAP command switches control between the LOG window and the SOURCE window when the debugger is running. When you begin a debugging session, the LOG window becomes active by default. While the DATA step is still being executed, the SWAP command enables you to switch control between the SOURCE and LOG window so that you can scroll and view the text of the program and also continue monitoring the program execution. You must enter the SWAP command from a window command line, from a menu, or with a function key.

TRACE

Controls whether the debugger displays a continuous record of the DATA step execution

Category: Manipulating Debugging Requests
Alias: T
Default: OFF

Syntax
TRACE <ON | OFF>

Without Arguments
TRACE displays the current status of the TRACE command.
**Arguments**

**ON**
- Prepares for the debugger to display a continuous record of DATA step execution. The next statement that resumes DATA step execution (such as GO) records all actions taken during DATA step execution in the DEBUGGER LOG window.

**OFF**
- Stops the display.

**Examples**

- Determine whether TRACE is ON or OFF:
  
  ```
  trace
  ```

- Prepare to display a record of debugger execution:
  
  ```
  trace on
  ```

---

**WATCH**

**Suspends execution when the value of a specified variable changes**

**Category:** Manipulating Debugging Requests

**Alias:** W

**Syntax**

```
WATCH variable(s)
```

**Arguments**

- **variable**
  - Specifies a DATA step variable.

**Details**

The WATCH command specifies a variable to monitor and suspends program execution when its value changes.

- Each time the value of a watched variable changes, the debugger:
  - Suspends execution
  - Displays the line number where execution has been suspended
  - Displays the variable's old value
  - Displays the variable's new value
  - Returns control to the user and displays the > prompt.

**Examples**

- Monitor the variable DIVISOR for value changes:
  
  ```
  w divisor
  ```
%DS2CSV Macro

Converts SAS data sets to comma-separated value (CSV) files

Valid: in a DATA step

Syntax

%DS2CSV(argument=value, argument=value,...)

Arguments That Affect Input/Output

csvfile=external-filename
specifies the name of the CSV file where the formatted output is to be written. If the file that you specify does not exist, then it is created for you.

Note: Do not use the CSVFILE argument if you use the CSVFREF argument. △

csvfref=fileref
specifies the SAS fileref that points to the location of the CSV file where the formatted output is to be written. If the file that you specify does not exist, then it is created for you.

Note: Do not use the CSVFREF argument if you use the CSVFILE argument. △

openmode=REPLACE|APPEND
indicates whether the new CSV output overwrites the information that is currently in the specified file or if the new output is appended to the end of the existing file. The default value is REPLACE. If you do not want to replace the current contents, then specify OPENMODE=APPEND to add your new CSV-formatted output to the end of an existing file.

Note: OPENMODE=APPEND is not valid if you are writing your resulting output to a partitioned data set (PDS) on z/OS. △
Arguments That Affect MIME/HTTP Headers

For more information about MIME and HTTP headers, refer to the Internet Request for Comments (RFC) documents RFC 1521 (http://asg.web.cmu.edu/rfc/rfc1521.html) and RFC 1945 (http://asg.web.cmu.edu/rfc/rfc1945.html), respectively.

\[\text{conttype} = Y \mid N\]
indicates whether to write a content type header. This header is written by default.

**Restriction:** This argument is valid only when \(\text{RUNMODE}=S\).

\[\text{contdisp} = Y \mid N\]
indicates whether to write a content disposition header. This header is written by default.

**Note:** If you specify \(\text{CONTDISP}=N\), then the \(\text{SAVEFILE}\) argument is ignored.

**Restriction:** This argument is valid only when \(\text{RUNMODE}=S\).

\[\text{mimehdr1} = \text{MIME/HTTP-header}\]
specifies the text that is to be used for the first MIME or HTTP header that is written. This header is written after the content type and disposition headers. By default, nothing is written for this header.

**Restriction:** This argument is valid only when \(\text{RUNMODE}=S\).

\[\text{mimehdr2} = \text{MIME/HTTP-header}\]
specifies the text that is to be used for the second MIME or HTTP header that is written. This header is written after the content type and disposition headers. By default, nothing is written for this header.

**Restriction:** This argument is valid only when \(\text{RUNMODE}=S\).

\[\text{mimehdr3} = \text{MIME/HTTP-header}\]
specifies the text that is to be used for the third MIME or HTTP header that is written when \(\text{RUNMODE}=S\) is specified. This header is written after the content type and disposition headers. By default, nothing is written for this header.

**Restriction:** This argument is valid only when \(\text{RUNMODE}=S\).

\[\text{mimehdr4} = \text{MIME/HTTP-header}\]
specifies the text that is to be used for the fourth MIME or HTTP header that is written. This header is written after the content type and disposition headers. By default, nothing is written for this header.

**Restriction:** This argument is valid only when \(\text{RUNMODE}=S\).

\[\text{mimehdr5} = \text{MIME/HTTP-header}\]
specifies the text that is to be used for the fifth MIME or HTTP header that is written. This header is written after the content type and disposition headers. By default, nothing is written for this header.

\[\text{runmode} = S \mid B\]
specifies whether you are running the \%DS2CSV macro in batch or server mode. The default setting for this argument is \(\text{RUNMODE}=S\).

- **Server mode** (\(\text{RUNMODE}=S\)) is used with Application Dispatcher programs and streaming output stored processes. Server mode causes DS2CSV to generate appropriate MIME or HTTP headers. For more information about Application Dispatcher, refer to the Application Dispatcher documentation at [http://support.sas.com/rnd/web/intrnet/stock/stock73.html](http://support.sas.com/rnd/web/intrnet/stock/stock73.html).
- **Batch mode** (\(\text{RUNMODE}=B\)) means that you are submitting the DS2CSV macro in the SAS Program Editor or that you included it in a SAS program.
Note: No HTTP headers are written when you specify batch mode. △

Restriction: RUNMODE=S is valid only when used within the SAS/IntrNet and Stored Process servers.

savefile=filename
specifies the filename to display in the Web browser’s Save As dialog box. The default value is the name of the data set plus “.csv”.

Note: This argument is ignored if CONTDISP=N is specified. △
Restriction: This argument is valid only when RUNMODE=S.

Arguments That Affect CSV Creation

colhead=Y | N
indicates whether to include column headings in the CSV file. The column headings that are used depend on the setting of the LABELS argument. By default, column headings are included as the first record of the CSV file.

data=SAS-data-set-name
specifies the SAS data set that contains the data that you want to convert into a CSV file. This argument is required. However, if you omit the data set name, DS2CSV attempts to use the most recently created SAS data set.

formats=Y | N
indicates whether to apply the data set’s defined variable formats to the values in the CSV file. By default, all formats are applied to values before they are added to the CSV file. The formats must be stored in the data set in order for them to be applied.

labels=Y | N
indicates whether to apply the data set’s defined variable formats to the values in the CSV file. By default, all formats are applied to values before they are added to the CSV file. The formats must be stored in the data set in order for them to be applied.

pw=password
specifies the password that is needed to access a password-protected data set. This argument is required if the data set has a READ or PW password. (You do not need to specify this argument if the data set has only WRITE or ALTER passwords.)

sepchar=separator-character
specifies the character that is used for the separator character. Specify the two-character hexadecimal code for the character or omit this argument to get the default setting. The default settings are 2C for ASCII systems and 6B for EBCDIC systems. (These settings represent commas (,) on their respective systems.)

var=var1 var2 ...
specifies the variables that are to be included in the CSV file and the order in which they should be included. To include all of the variables in the data set, do not specify this argument. If you want to include only a subset of the variables, then list each variable name and use single blank spaces to separate the variables. Do not use a comma in the list of variable names.

where=where-expression
specifies the variables that are to be included in the CSV file and the order in which they should be included. To include all of the variables in the data set, do not specify this argument. If you want to include only a subset of the variables, then list each variable name and use single blank spaces to separate the variables. Do not use a comma in the list of variable names.
Details
The DS2CSV macro converts SAS data sets to comma-separated value (CSV) files. You can specify the hex code for the separator character if you want to create some other type of output file (for example, a tab-separated value file).

Example
The following example uses the %DS2CSV macro to convert the SASHELP.RETAIL data set to a comma-separated value file:

```sas
data sashelp.retail;
%ds2csv (data=sashelp.retail, runmode=b, csvfile=c:\temp\retail.csv);
run;
```
Recommended Reading

Here is the recommended reading list for this title:

- An Array of Challenges—Test Your SAS Skills
- Base SAS Glossary
- Base SAS Procedures Guide
- Cody’s Data Cleaning Techniques Using SAS Software
- Combining and Modifying SAS Data Sets: Examples
- Debugging SAS Programs: A Handbook of Tools and Techniques
- Health Care Data and the SAS System
- The Little SAS Book: A Primer
- Output Delivery System: The Basics
- Quick Results with the Output Delivery System
- SAS Companion for OpenVMS Alpha
- SAS Companion for UNIX Environments
- SAS Companion for Windows
- SAS Companion for z/OS
- SAS Guide to Report Writing: Examples
- SAS Language Reference: Concepts
- SAS Metadata LIBNAME Engine: User’s Guide
- SAS Programming by Example
- SAS Scalable Performance Data Engine: Reference
- The SAS Workbook
- SAS XML LIBNAME Engine User’s Guide
- Step-by-Step Programming with Base SAS Software
- Using the SAS Windowing Environment: A Quick Tutorial

For a complete list of SAS publications, see the current SAS Publishing Catalog. To order the most current publications or to receive a free copy of the catalog, contact a
SAS representative at

SAS Publishing Sales
SAS Campus Drive
Cary, NC 27513
Telephone: (800) 727-3228*
Fax: (919) 677-8166
E-mail: sasbook@sas.com
Web address: support.sas.com/pubs
* For other SAS Institute business, call (919) 677-8000.

Customers outside the United States should contact their local SAS office.
Index

& (ampersand) format modifier, definition 1272
@ (at sign) line-hold specifier, PUT statement 1352
@@ (at signs) line-hold specifier, PUT statement 1352
: (colon) format modifier, definition 1272
; (semicolon), in data lines 1120, 1135
~ (tilde) format modifier, definition 1272

A
ABEND argument, ABORT statement 1101
ABORT statement 1100
ABBEND argument, ABORT statement 1101
ABS function 294
absolute values 294
access methods
URL access method 1206
ACCESS= option
LIBNAME statement 1287
ACCESS=READONLY option
CATNAME statement 1121
ADD method 1638
ADDR function 294
ADDRLONG function 296
aggregate storage location filerefs for 1176
AIRY function 297
derivative of 477
value of 299
alignment of output
_ALL_ argument 1492
FILENAME statement 1173
LIBNAME statement 1287
_ALL_ CLEAR option
CATNAME statement 1121
_ALL_ LIST option
CATNAME statement alphabet characters
ALTER= data set option alter passwords
@ (at sign) line-hold specifier 8
ampersand (&) format modifier 1272
ampersand format modifier 1272
anonymous FTP login 1199
ANYNUM function 298
ANYALPHA function 300
ANYCNTRL function 302
ANYDIGIT function 303
ANYDDTDTEw. informat 966
ANYDTDTMw. informat 970
ANYDITTMew. informat 971
ANYFIRST function 304
ANYGRAPH function 306
ANYLOWER function 308
ANYNAME function 309
ANYPRINT function 311
ANYPUNCT function 313
ANYSPACE function 314
ANYUPPER function 316
ANYXDIGIT function 317
APPLETLOC= system option applications
ARM API objects arc cosine 319
arc sine 320
arc tangent 320
ARCCOS function arguments converting to lowercase 641
converting words to proper case 737
data type, returning 912
difference between nthlag 496
date extracting substrings 840
format decimal values, returning 887
format names, returning 889
format width, returning 900
informat decimal values, returning 902
informat names, returning 904
informat width, returning 906
resolving 319
returning length of 626
returning arithmetic mean 644
ARM agents 1460
ARM API function calls 1054
ARM API objects 1056
ARM logs converting 1069
processing 1078
ARM macros by category call schemes conditional execution
defining correlators in defining user metrics enabling execution 1065
error messages 1067
execution with SCL 1064
ID management with overview 1054
post-processing macros setting macro environment variables with 1055
ARM subsystems
ARMAGENT= system option 1460
%ARMCONV macro 1069
ARM_DSIO subsystem 1464
%ARMEND macro 1071
_ARMEXEC macro variable 1074
%ARMGETID macro 1072
ARM_OLAP_SESSION subsystem 1466
%ARMINIT macro 1076
ARM_PROC subsystem 1466
%ARMJOIN macro 1072
ARMLOC= system option 1461
ARM_OOLAP_SESSION subsystem 1470
%ARMPROC macro 1078
ARM_PROC subsystem 1470
%ARMSTOP macro 1079
%ARMSTRK macro 1081
ARMSUBSYS= system option 1462
ARM subsystems 1464
ARM_DSIO subsystem 1464
ARM_OLAP_SESSION subsystem 1466
ARM_PROC subsystem examples 1470
%ARMUPDT macro array reference, explicit 1084
compared to ARRAY statement array reference statement 1107
ARRAY statement 1103
compared to array reference, explicit 1109
arrays 498
defining elements in describing elements to process finding contents 894
finding dimensions 498
finding values in identifying 880
lower bounds 624
upper bounds of writing to 1333
AR SIN function 320
ASCII characters, returning a string of 344
by number 334
number of ASCII data
  converting character data to native format 93
  converting to native format 94
  ASCIIw. format 93
  ASCIIw. informat 94
  asymmetric I/O 1477
  asymmetric spelling differences 828
  AUTOMACROS option, WINDOW statement 1110
  AUTOSAVELOC= system option 1477
  AUTOSAVE= option, WINDOW statement 1477
  AUTOSAVELOC= system option 1477
  AUTOSAVE= option, WINDOW statement 1477
  average 644
  ATTACH= option 1479
  batch mode 1479
  batch processing 1479
  error handling 1527
  BATCH system option 1479
  BCC= option
  FILENAME statement 1185
  BELL argument, DISPLAY statement 1185
  Bernoulli distributions 417
  cumulative distribution functions 417
  probability density functions 696
  Bessel function, returning value of
  BESTw. format 1113
  compared to Dw. format 983
  beta distributions 630
  cumulative distribution functions 630
  probability density functions 697
  quantiles 330
  BETA function 329
  BETAINV function 329
  big endian platforms 75
  byte ordering 935
  big endian platforms, byte ordering on
  bin specification, for printed output 1582
  bin specification, for sending paper to
  printer 1584
  binary data
  converting character data to 90
  converting numeric data to 114
  binary data, converting to character
  binary integers 949
  binary option
  FILENAME statement, FTP access
  method 1194
  binary zeros, converting to blanks 953
  $SBINARYw. format 94
  $BINARYw. format 94
  $BINARYw. informat 949
  $BINARYw.d informat 973
  $BINDING= system option 1480
  $BYLINE system option 1487
  $BYGROUP system option 334
  $BYTE function 334
  byte ordering 935
  for integer binary data 75
  BZw.d informat 975
  BZW.D informat 975
  compared to w.d informat 1043
  converting to zeros 975
  trimming trailing 864, 865
  BLKSIZE= option
  FILE statement 1159
  BUFNo= system option 1482
  BUFSIZE= data set option 1482
  BUFSIZE= system option 1482
  FILENAME statement, FTP access
  method 1194
  FILENAME statement, SOCKET access 1202
  BLKSIZE= option
  FILE statement 1159
  BUFNo= system option 1482
  BUFSIZE= data set option 1482
  BUFSIZE= system option 1482
  FILENAME statement, FTP access
  method 1194
  FILENAME statement, SOCKET access 1202
  BLSHIFT function 331
  BNOT function 332
  bookmarks 548
  finding 716
  setting 548
  BOR function 332
  BOTTOMMARGIN= system option 1481
  BREAK command
  DATA step debugger 1680
  BRSHIPT function 333
  buffers number for data sets 1482
  number for processing data sets 1482
  page buffers for catalogs 12
  page size and permanent page size
  size of 1484
  viewer buffer size 38
  BY lines
  printing 1486
  BY statement 1486
  BY values
  duplicates, MODIFY statement 1313
  BY variables
  customizing titles with 1412
  BYE command, compared to ENDSAS statement
  1155
  compared with DSNFEK system option 1518
  BYERR system option 1485
  BYGROUP system option 1487
  BYTE function 334
  byte ordering 935
  for integer binary data 75
  BZW.D informat 975
  compared to w.d informat 1043
  C
  CALCULATE command
  DATA step debugger 1682
  CALL ALLPERM routine 335
  CALL CATS routine 337
  CALL CATT routine 339
  CALL CATX routine 240
  CALL COMPCOST routine 342
  CALL LOGISTIC routine 346
column-binary data, reading down a column punch-card code 1018

column-binary files, reading column input 1253, 1269

COLUMN= option
FILE statement 1159
INFILE statement 1225
column output 1349, 1361
column pointer controls
INPUT statement 1250
PUT statement 1346
columns
two-column page format 1168
COLUMNS= argument, WINDOW statement 1427
COMB function 442

See permutations, computing combinations, computing 442
column-binary files, reading
compressing character strings 458
tab characters 458
lowercase letters 458
column-binary data, reading
compressing data sets compressing observations concatenating catalogs
CATNAME statement implicitly logically concatenated catalogs
nested catalog concatenation rules for
concatenating data libraries logically
concatenating data sets
SET statement for confidence intervals, computing
CONSTANT function constants, calculating
double-precision numbers, largest 461
double-precision numbers, smallest 461
Euler constant 460
exact integer 460
machine precision 462
natural base 459
overview 459
CONTENT_TYPE= option
FILENAME statement 1184
CONTINUE argument, DM statement 1144
CONTINUE statement compared to LEAVE statement control characters
searching strings for convexity, for enumerated cashflow stream
convexity, for periodic cashflow stream
CONV function 463
CONVX function 462
copies, specifying number of
COPIES= option
corrected sum of squares correlators in ARM macros 1062
COS function 465
COSH function 466
cosine 466
COUNT function 466
EXP function 467
COUNTC function 468
CPU function 469
cumulative distribution functions
Bernoulli distribution 417
beta distribution 417
binomial distribution 418
Cauchy distribution 419
chi-squared distribution 419
exponential distribution 420
F distribution 420
gamma distribution 421
geometric distribution 421
hypergeometric distribution 422
Laplace distribution 423
logistic distribution 423
lognormal distribution 423
negative binomial distribution 423
normal distribution 424
Pareto distribution 425
Poisson distribution 426
T distribution 426
t uniform distribution 427
Wald (Inverse Gaussian) distribution 427
Weibull distribution 428
CVP function 471
CVPBYTES= option
LIBNAME statement 1288
CYPENGINE= option
LIBNAME statement 1288
CVPMULTIPLIER= option
LIBNAME statement 1288

D

DACDB function 472
DACDBSBL function 473
DACCSL function 474
DACCSYD function 475
DACCTAB function 476
DAIRY function 477
damaged data sets
1511
message
1291
writing attributes to log
data lines
as card images 1489
length of sequence field reading 1120, 1135

DAMAGED data sets or catalogs
16

Data conversion formats and 77
data files encryption 18
data libraries associating librefs with concatenating, logically
disassociating librefs from transporting
verifying existence of members as card images 1489
length of sequence field reading 1120, 1135

Data Set Data Vector (DDV), reading observations into 523, 524
Data Set Names, returning
508
by category 5
input data sets and MODIFY statement with 1316
output data sets
7
system option interactions 1448
data set pointer, positioning at start of data set 788
data sets
buffer number for processing 11
buffer page size 12
buffer size 1484
combining 1463
compressing 14

DAMAGED data sets 16

DAMAGED data sets 16
concatenating contributing to current observation 1403
converting to CSV files 1695
converted damaged 10
defining indexes at creation 129
dropping variables 127
extracting zip codes from generations for 49
importing transport data sets 23
input/output processing information 1464
interleaving keeping variables 30
labels for number of buffers numeric attribute for one-to-one reading permanently storing, one-level names reading observations reading observations, more than once replacing like-named replacing when empty reusing free space sharing access levels sorting special structures verifying existence of DATA statement arguments 1127 DEBUG option 1688 examples 1131 keywords allowed in stored compiled DATA step programs syntax 1130 DATA step assigning data to macro variables 401 creating component objects 1132 declaring component objects 1136 instantiating component objects 1136 linking SAS data set variables 393, 401 MODIFY statement in starting 1127 stopping 1138, 1407 stored compiled programs terminating 1100 DATA step component object interface DATA step component objects 1635 dot notation 1636 DATA step debugger assigning commands to ENTER key assigning commands to function keys assigning new variable values 1690 commands by category 1678 continuous record of DATA step execution 1692 customizing commands with macros DATA step generated by macros DEBUG option 1667 debugger sessions 1665 debugging, defined debugging DO loops 1677 deleting breakpoints 1683 deleting watch status description of 1664 displaying variable attributes 1684 displaying variable values 1685 entering commands 1665 evaluating expressions 1682 examples 1667 executing statements one at a time expressions and formats and formatted variable values help on commands jumping to program line list of commands listing items macro facility with macros as debugging tools quitting restarting suspended programs resuming DATA step execution starting DATA step execution suspending execution switching window control windows DATA step functions, within macro functions 238 DATA step programs stored compiled, executing 1137 DATA step programs, retrieving source code from 1114 DATA step statements declarative executable 1090 global, by category 1096 global, definition 1095 DATA step views creating 1130 describing 1130 non-sequential processing 52 retrieving source code from spill file for view buffer size 38 data type, returning data values, reading data views verifying existence of DATALINES argument INFILE statement 1225, 1236 DATALINES statement compared to DATALINES4 statement 1133 length of data on lines following 1135 DATALINES4 statement 1135 DATASTMTCHK= system option DATDIF function 1047 date and time values date, mm/dd/yyyy 163, 164 Julian day of the year 160
Julian dates, converting to SAS values 479 Julian dates, from SAS date values minute values, returning 647
month values, returning 480 returns second value, returning 821 returns time, extracting from datetime values 856 returns time, returning current 840 returns time intervals, extracting integer values of time values, creating 590 year quarter, returning 264 year quarter, returning date value from year value, returning 917 date/time values
year 2000 accommodation year cutoff 1631 date/time values, reading date, yymm 1046 date, yymmm 1046 date values, ddmmmmymy 979 date values, dddmmmyy hh:mm:ss.ss 981 date values, dddmmmmyy hh:mm:ss.ss 979 date values, dddmmmmyy hh:mm:ss.ss 981 dates, mmddyy 982 dates, mmddyyyy 984 dates, yymmmdd 1044 dates, yyyyymmmdd 1044 IBM mainframes, RMF records 1016 IBM mainframes, SMF records 1035 Julian dates 992 month and year values 995 RMF records 1007 SMF records 1007 time, hh:mm:ss.ss 1037 time, hh:mm:ss ss 1037 TIME MIC values 997 time-of-day stamp 1039 time values, IBM mainframe timer units 1044 year quarter 1047 date/time values, writing month abbreviation 232 Roman numerals, year and quarter 237 year and month, with separator character 227 year and quarter, Roman numerals with separator character 237 year and quarter, with separator character 235 date values aligning output as day of month as day-of-week name day of week day of week and date dd month-name yyyy mmdddy 169 mmmmy 167 mmmmyyy 172 mmyy 167 year and month dd, yyyy 221 month of the year name of month quarter of the year year
| I/O control | MODIFY statement | 1324 |
| IBM mainframe format | integer binary values in packed decimal data | 195 |
| | integer binary values in real binary data | 198 |
| | unsigned integer binary values in unsigned packed decimal data | 202 |
| | unsigned integer binary values in unsigned decimal data in writing numeric data | 203 |
| | zoned decimal data in zoned decimal leading-sign data | 204 |
| | zoned decimal separate leading-sign data in | 206 |
| | zoned decimal separate trailing-sign data in | 207 |
| IBM packed decimal data, reading | encoding | 002 |
| IBfw.d format | 154 |
| IBfw.d informat | 998 |
| IBUSIZE= option | 1538 |
| IBw.d format | 155 |
| IBw.d informat | 998 |
| | compared to S700FIBw.d informat | 1021 |
| ICOLUMN= argument, WINDOW statement | 1427 |
| IDXXNAME= data set option | 25 |
| IDXXWHERE= data set option | 26 |
| IEEE floating-point values reading | 991 |
| IEEEfw.d format | 159 |
| IF, THEN/ELSE statements | 1216 |
| IF statement, subsetting | 1215 |
| | compared to DELETE statement | 1140 |
| IFC function | 584 |
| IFN function images | 886 |
| IML procedure | MODULE�N function in IML procedure | 350 |
| | importing transport data sets | 1200 |
| IN= data set option | 28 |
| INCLUDE command, compared to %INCLUDE statement | 1222 |
| %INCLUDE statement | 1218 |
| catalog entries with examples | 1222 |
| external files, including (example) | 1222 |
| including previously submitted lines (example) | 1222 |
| keyboard input, including (example) | 1222 |
| processing large amounts of data | 1397 |
| sources of data for with several entries in a single catalog (example) | 1222 |
| including programming statements and data lines | 1258 |
| incrementing values | 602 |
| INDEX= data set option | 29 |
| INDEX function defined to INDEXC function | 590 |
| INDEXC function indexed values direct access by indexes | 1313 |
| defining at data set creation | 29 |
| duplicate values | 1313, 1322 |
| optimizing WHERE expressions overriding | 26 |
| shift indexes | 604 |
| WHERE expressions and INDEXW function | 26 |
| INENCODING= option | 591 |
| LIBNAME statement | 1288 |
| _INFILE_ option | 1346 |
| PUT statement | 1346 |
| INFILE statement | 1231, 1242, 1243 |
| INFILE statement | 1254 |
| compared to INPUT statement | 1259 |
| comparisons | 1236 |
| DBMS specifications | 1232 |
| delimited data, reading delimiters | 1236 |
| details | 1232 |
| encoding specification examples | 1236 |
| input buffer, accessing contents input buffer, working with data | 1241 |
| missing values, list input | 1232 |
| multiple input files | 1233 |
| operating environment options options | 1225 |
| pointer location | 1241 |
| reading long instream data records | 1235 |
| reading past the end of a line short records | 1237 |
| truncating copied records updating external files in place | 1240 |
| variable-length records, reading | 1233 |
| variable-length records, scanning | 1238 |
| informat decimal values, returning | 897 |
| arguments | 896 |
| informat names, returning | 898 |
| arguments | 900 |
| variables | 908 |
| INFORMAT= option | 1112 |
| ATTRIB statement | 1245 |
| INFORMAT statement | 1245 |
| specifying informats with informat width, returning | 913 |
| arguments | 902 |
| variables | 901 |
| informats | 932, 941 |
| ambiguous data | 1507 |
| associating with variables | 1111, 1245 |
| byte ordering | 935 |
| categories of integer binary notation | 941 |
| name length | 936 |
| permanent | 934 |
| reading results of expressions | 592 |
| reading unaligned data with returning | 1244 |
| specifying | 933 |
| specifying, with ATTRIB statement specifying, with INFORMAT statement specifying, with INPUT function specifying | 933 |
| specifying, with INFILE statement specifying at run time syntax | 933 |
| temporary | 934 |
| user-defined | 934 |
| INITCMD system option | 1540 |
| INITSTMT= system option input | 1541 |
| as card images | 1489 |
| assigning to variables | 1248 |
| column | 1253, 1263 |
| describing format of end-of-data indicator | 1245 |
| error detection levels | 1330 |
| formatted | 1509 |
| invalid data list | 1253 |
| list input | 1269 |
| listing for current session | 1298 |
| logging | 1298 |
| missing records | 1304 |
| missing values | 1304 |
| named | 1254, 1276 |
| resynchronizing | 1304 |
| sequence field, length of numeric portion | 1599 |
| uppercasing | 1488 |
| input buffer accessing, for multiple files | 1243 |
| accessing contents | 1233 |
| working with data in | 1241 |
| input column | 1266 |
| input data | 1235 |
| reading past the end of a line | 1235 |
| input data sets data set options with | 6 |
| redirecting | 1376 |
| input files | 1240 |
| reading multiple files | 1233, 1239 |
| truncating copied records | 1240 |
| INPUT function specifying informats with | 933 |
null
remote files creating 1199
FTP access method 1199
remote SAS sessions 1512
REMOVE statement 1547
compared to OUTPUT statement 1552
compared to REMOVE statement 1537
compared to REPLACE statement 1538
RENAME= data set option 46
compared to RENAME statement 1379
derror detection for input data sets 1509
RENAME= data step option 1579
derror detection for output data sets 1510
RENAME statement 1379
ererror detection for output data sets 1510
renaming variables 60
REPEAT function 786
REPEMTY= data set option 47
REPEMTY= option LIBNAME statement 1289
REPLACE statement 1381
compared to REPLACE= system option 1591
REPLACE method 1660
REPLACE statement 1381
compared to OUTPUT statement 1532
compared to REMOVE statement 1377
REPLACE system option 1591
REPLYTO= option FILENAME statement 1185
REVERS function 786
RESOLVE function 786
resolving arguments 344
assigning at startup 1552
Results window invoking 1513
RETAIN statement 1383
compared to KEEP statement 1279
compared to SUM statement 1409
RETURN statement 1388
compared to GO TO statement 1213
REUSE data set option 1591
compared with REUSE= system option 1592
REUSE= system option 1591
SREVERS format 102
REVERSE function 786
reverse order character data SREVERS format 103
REWIND function 788
RHELP option
FILENAME statement, FTP access method 1197
RIGHT function 789
RIGHTMARGIN= system option 1593
RMF records, reading duration intervals 1015
RMFDURw. informat 1015
RMFDURw. informat 1016
compared to RMFDURw. informat 1015
RMS function 790
roman numerals 193
root mean square 790
ROUND function 790
ROUND function rounding 790
ROUNDDZ function 799
ROWS= argument, WINDOW statement 1428
ROWw.d informat 1018
RSASUSER system option 1594
RSTAT option FILENAME statement, FTP access method 1197
RUN statement 1389
%RUN statement 1390
RXCHANGE CALL routine 385
RXFREE CALL routine 387
RXMATCH function 801
RXPARSE function 802
RXSUBSTR CALL routine 387
S= system option 1594
compared with S2= system option 1220
S2= argument, INCLUDE statement 1220
S2= system option 1596
compared with S= system option 1595
S370FWd format 194
S370FWd informat 1019
S370FBDUw.d format 197
S370FBDUw.d informat 1022
S370FPDw.d format 198
S370FPDw.d informat 1024
compared to S370FPDUw.d informat 1025
S370FPDUw.d format 199
S370FPDUw.d informat 1025
S370FDw.d format 197
S370FDw.d informat 1024
compared to S370FDUw.d informat 1025
SORT= system option 1360
sas catalog entries, verifying existence 432
SAS call entries, verifying existence 432
SAS catalogs verifying existence 432
SASCONNECT software remote session ability 1512
SAS data libraries damaged data sets or catalogs 1511
default access method 1525
default permanent, specifying listing details 1506
paths, returning 693
SAS data sets 322
character attributes, returning 322
character variables, returning values of closing 436
compressing on output 1507
deleting observations 1501
SAS/AF software most recently created, specifying not found 1517
note markers, returning 507
numeric variables, returning values of opening 690
replacing 1591
reusing free space 1591
setting data set pointer to start of 788
variable data type, returning 882
variable labels, returning 874
variable length, returning 878
variable names, returning 878
variable position, returning 878
writing to 1341
SAS/FSP software, multiple-environment support 1569
SAS functions 252
SAS/GRAPH software displaying output in GRAPH window 1537
terminal device driver, specifying 1508
SAS informats 932
SAS jobs, terminating 1100
SAS log
AUTOEXEC input 1520
date and time, printing detail level of messages 1506
catalogs 1568
error messages 1588
logging input 1298
news file 1570
notes 1571
secondary source statements skipping to new page 1344
source statements 1605
writing hardware information to 1505
SAS OPTIONS window, compared to OPTIONS statement 1340
SAS print file, initializing 1588
SAS procedure output aligning 1492
SAS procedures labels with variables 1543
SAS sessions DCOM/CORBA server mode executing statements at termination 1572
issuing operating-system commands 1614
terminating 1100
SAS/SHARE MODIFY statement and 1316
SAS statements 1089
executing at startup 1541
writing to utility data set in WORK data library

SAS system options

CBUFNO= 1491
CENTER 1492
changing values of 1340
CHARCODE 1493
CLEANUP 1494
COLLATE 1499
COLORPRINTING 1500
COMPRESS= 1501
COPIES= 1503
CPUID 1505
DATASTMTCHK= 1505
DATE 1506
DETAILS 1508
DEVICE= 1508
DKRICOND= 1509
DKROCOND= 1510
DLDMGACTION= 1511
DSNFERR 1517
DUPLEX 1519
ECHOAUTO 1520
ENGINE= 1525
ERRORABEND 1526
ERRORCHECK= 1527
FIRSTOBS= 1530
FMTERR 1531
FMTSEARCH= 1532
FORMCHAR= 1534
FORMDLIM= 1534
FORMS= 1535
GWINDOW 1537
INITSTMT= 1541
INV ALIDDA TA= 1542
LABEL 1543
_LAST_= 1544
LEFTMARGIN= 1544
MERGENOBY 1552
MISSING= 1566
MSLEVEL= 1568
MULTENV APPL 1569
NEWS= 1570
NOTES 1571
NUMBER= 1571
ORIENTATION= 1578
OVP 1579
PAGENO= 1580
PAGESIZE= 1581
PARDEST= 1582
PAPERSOURCE= 1584
PAPERTYPE= 1584
PARM= 1585
PARMCARDS= 1586
PRINTINIT 1588
PRINTMSGLIST 1588
REPLACE 1591
REUSE= 1591
RIGHTMARGIN= 1593
RSASUSER 1594
S= 1594
S2= 1596
SASHELP= 1597
SASUSER= 1598
SEQ= 1599
SKIP= 1600
SOLUTIONS 1601
SORTDUP= 1602
SOURCE 1605
SOURCE2 1605
SPool 1606
STARTLIB 1607
SUMSIZE= 1608
TERMINAL= 1613
TOPMARGIN= 1617
TRUNCATES= 1618
USER= 1620
VALIDV ARNAME= 1626
VNFERR 1627
WORK= 1628
WORKINIT 1629
YEARCUTOFF= 1631

SAS windowing environment

invoking 1513
syntax checking 1516

SASFILE statement 1391
SASHELP library, location of 1597
SASHELP= system option 1597
SASUSER library
access control name, specifying 1598
SASUSER= system option 1598
SAVING function 815
SCAN function 816
SCANOVER option
INFILE statement 1230, 1238
SECOND function 821
SECOND function
interleaving data sets 1403
merging observations 1404
one-to-one reading option 1400
reading subsets 1405
table-lookup transcoded variables with
SETINIT system option 1600
SHAREBUFFERS option
INFILE statement 1230, 1240
SHR records
reading data and time values of
SHRSTAMPw. informat 1034
SIGN function signs, returning 822
SIN function sine 822
SINH function site license information
altering 1600
skewness 824
SKEWNESS function 824
SKIP statement 1406
SLEEP function 824
SMALLEST function 826
SMFSTAMPw. informat
SMTP access method
FILENAME statement 1184
Social Security numbers
SOCKET access method
FILENAME statement 1202
SOCKET argument
FILENAME statement, SOCKET access
SOCKET argument
FILENAME statement, SOCKET access
text
SOFTMAX function 395
SOLUTIONS folder 1601
SOLUTIONS menu choice 1601
SOLUTIONS system option
SORT procedure
error messages 1485
memory for 1604
NODUP option 1602
SORTDUP= system option 1602
SORTEDBY= data set option
SOREQUALS system option 1602
SORT procedure
data sets 50
SORTSIZE= system option 1604
SOUNDEX function 827
SOURCE entries writing to 1180
TAMEAMPMw.d format 211
TIMEPART function 556
TIMEw. informat 1037
TIMEw. format 154, 209
compared to HHMMw.d format 154
TINV function 557
TITLE statement titles 1410
customizing with BY variables 1412
customizing with ODS 1412
TITLES option 1165
TNONCT function 858
TO= option
FILENAME statement 1184
TO statement, compared to LINK statement 1296
TOBSNO= data set option 60
TO statement, compared to MERGE statement 1308
TRANSPORT statement with 1248
TRRR function 607
true expressions 584, 586
TRUNC function 856
truncating copied records 1240
TRUNCOVER option
FILENAME statement 1231, 1238
TUw. informat 1041
two-column format 1168
TYPE= data set option 61

U
unaligned data 1222
UNBUFFERED option
FILENAME statement 1231
uncorrected sum of squares 871
uniform distributions 384, 427
cumulative distribution functions 427
probability density functions 706
random numbers 384, 785, 867
UNIFORM function 867
UNIQUE option
SET statement 1402
MODIFY statement 1312
universal printers
fileres for 1175
Universal Printing enabling 1619
printer designation 1587
Universal Unique Identifier (UUID) 871
IBM mainframe format 197
unsigned integer binary data 1022
IBM mainframe format 1022
unsigned packed decimal data 1025
IBM mainframe format 188
signed packed decimal format 1012
unsigned packed decimal format 188
IBM mainframe format 208
unsigned zoned decimal data 1033
IBM mainframe format 1033
UPCASE function 868
SUPCASEw. format 108
SUPCASEw. informat 964
UPDATE statement 1411
compared to MERGE statement 1308
UPDATEMODE= argument
UPDATE statement 1417
UPDATEMODE= option
MODIFY statement 1312
uppercase 868
converting character data to
converting character expressions to
reading data as 1488
translated input to
UPCASE function 868
SUPCASEw. informat 964
writing character data in
URL access method
accessing files at a Web site 1207
FILENAME statement 1206
reading part of a URL file 1208
user ID and password
URL argument
FILENAME statement, URL access method 1206
URLENCODE function 869
URLDECODEN function 870
URLS decoding 869
encoding 870
escape syntax 869, 870
user-defined formats 74
USER= option
FILENAME statement, FTP access method 1198
USER= system option 1620
US$ function 877
UTILLOC= system option 1620
UUID Generator Daemon host and port 1622
number of UUIDs to acquire 1621
UUID (Universal Unique Identifier) 871
UUIDCOUNT= system option 1621
UUIDGEN function 871
UUIDGENHOST= system option 1622

V
V6CREATEUPDATE= system option 1624
VALIDFMTNAME= system option 1624
VALIDVARNAME= system option values
signs, returning 822
VAR function 872
VAFRMFT function 873
variable-length records 1238
scanning for character string variables 295, 345, 405
assigning input to 1248
associating formats with
associating informat with
associating labels with
associating length with
character, returning values of
data type, returning 882
dropping 17
_ERROR_. setting 1156
format decimal values, returning 886
format names, returning 888
format not found 1531
format width, returning 890
format decimal values, returning 896
format names, returning 898
format width, returning 901
keeping 50
labeling 1280
labels, assigning labels, returning labels with, in SAS procedures
length, returning 1543
length, specifying 1284
memory address of names, assigning 295
names, defining names, returning numeric, returning values of
operating system, returning 848
ZDw.d format 240
ZDw.d informat 1048, 1051
  compared to ZDVw.d 1051
  zeros, binary
  converting to blanks 953
ZDw.d informat 1048, 1051
  zeros, binary
  converting to blanks 953
ZIP codes 923
  converting to FIPS codes 923
  converting to postal codes 927
  converting to state names 924, 925
ZIPFIPS function 923
ZIPNAME function 924
  compared to ZIPNAMEL function 926
  compared to ZIPSTATE function 927
ZIPNAMEL function 925
  compared to ZIPNAME function 924
  compared to ZIPSTATE function 927
ZIPSTATE function 927
  compared to ZIPNAME function 924
  compared to ZIPNAMEL function 926
  compared to ZIPSTATE function 927
zoned decimal data 938
  defined 937
  formats and 77
  formats and informs for 80, 939
IBM mainframe format 203
  languages supporting 79, 938
  platforms supporting 70, 938
zoned decimal data, reading
  IBM mainframe format 1028
zoned decimal format 240
zoned decimal leading-sign data
  IBM mainframe format 204
zoned decimal leading-sign data, reading
  IBM mainframe format 1030
zoned decimal separate leading-sign data
  IBM mainframe format 206
zoned decimal separate trailing-sign data
  IBM mainframe format 207
zoned decimal separate trailing-sign data, reading
  IBM mainframe format 1032
zoned separate leading-sign data, reading
  IBM mainframe format 1031
Zw.d format 215, 239
  compared to w.d format 215
If you have comments or suggestions about *SAS 9.1.3 Language Reference: Dictionary, Volumes 1, 2, and 3*, please send them to us on a photocopy of this page, or send us electronic mail.

For comments about this book, please return the photocopy to

SAS Publishing  
SAS Campus Drive  
Cary, NC 27513  
E-mail: yourturn@sas.com

For suggestions about the software, please return the photocopy to

SAS Institute Inc.  
Technical Support Division  
SAS Campus Drive  
Cary, NC 27513  
E-mail: suggest@sas.com