CHAPTER

16

SAS Component Language Dictionary

ACTIVATE

Activates or grays either a Version 6 check box or radio box widget, or a station in a choice group

Category: Control or Field

Syntax

rc=**ACTIVATE**(*var-name,station*<,*row*>);

rc

contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

var-name

is the FRAME entry control or choice group to activate.

Type: Character

station

is the number of the station or item to be activated or grayed:

>0	is the number of the station to be activated.
<0	is the number of the station designated by the absolute value of the argument to be grayed instead of activated. Users cannot select a station that has been grayed.
0	indicates that no station is to be activated.

Type: Numeric

row

is the row number for a choice group that is in the scrollable section of an extended table. *Row* is valid for PROGRAM entries but not for FRAME entries. Specify *row* only when you want to activate a station from outside an extended table's getrow or putrow section. Do not specify *row* if you want to activate a station with the _getrow or _putrow method or from the getrow or putrow sections.

Type: Numeric

Details ACTIVATE works only for Version 6 widgets in FRAME entries and for choice groups in PROGRAM entries. Version 8 controls use attribute settings to implement this functionality. ACTIVATE is not valid in FSEDIT or FSVIEW programs.

The *station* value is the number of the check box or the item or button in a check box or choice group. For example, if your application has three fields named A, B, and C, and they all belong to the same radio box, then you can make the B field active by passing 2 for the *station* value (B is the second field).

ACTIVATE makes a check box or an item in a radio box or choice group active by assigning the station or item number to the associated window variable. ACTIVATE grays a check box or an item in a radio box or choice group when you assign the negative number of the station or item. When a window element is grayed, users cannot select it.

FRAME entry controls can also use the _activate method.

For linked action fields in choice groups, the action-type pair is considered one station. Linked action fields have the following form:

```
& &A_____    & &B_____     & &C_____
```

To make the linked action pair for B active, pass 2 for the value of *station*, not 4.

Examples

Example 1: Activating a Radio Box Station Activate the second station in the radio box HOBBY:

```
rc=activate('hobby',2);
```

Example 2: Activating a Station in an Extended Table Make the third station in the fourth row of an extended table the active station in the choice group LIST:

```
if (activate('list',3,4)) then
   do;
  ...SCL statements to handle the error condition...
   end;
```

See Also

"GRAY" on page 457

"ISACTIVE" on page 492

"ISGRAY" on page 493

"UNGRAY" on page 693

ALARM

Sounds an alarm on a device when the current window is refreshed or redisplayed.

Category: Utility

Syntax

ALARM;

Details The ALARM statement sounds the bell when the current window is refreshed or redisplayed. This statement works for devices that support sounds.

See Also

"CURSOR" on page 323

"FIELD" on page 394

APPEND

Appends a new row to a SAS table

Category: SAS Table

Syntax

sysrc=**APPEND**(*table-id*<,*option*>);

sysrc
contains the return code for the operation:

0	successful
≠0	an error or warning condition occurred

table-id
contains the identifier for the SAS table, which is returned by the OPEN function.
Type: Numeric

option
is one of the following options:

'NOSET'
appends a row with all column values set to missing, even if the SET routine has been called.

'NOINIT'
appends a row with the values currently in the Table Data Vector (TDV), even if the SET routine has not been called.
Type: Character

Details APPEND adds a row to the end of a SAS table. By default, the added row contains missing values. However, you can add a row with the current values in the TDV if SET has been called or if the NOINIT option is specified as the second argument. If SET has been called, the NOSET argument can still force APPEND to fill the row with missing values.

If the SET routine has not been called, you can PUTVARC and PUTVARN to specify values for each column in the TDV before calling APPEND with the NOINIT option. You can use INITROW to initialize the TDV to missing to prevent APPEND from writing bad data to a row values are not explicitly assigned to some columns through PUTVARC or PUTVARN.

Example

Add a row to the SAS table WORK.DATAONE, which has two columns, FNAME and SSN. Because SET is called, the values **ROBERT** and **999-99-9999** are written to the new row.

```
tableid=open('work.dataone','u');
call set(tableid);
fname='Fname';
ssn='999-99-9999';
if (append(tableid)) then do;
     _msg_=sysmsg();
```

If SET had not been called, then using the NOINIT option would produce the same results:

```
tableid=open('work.dataone','u');
fname='ROBERT';
ssn='999-99-9999';
call putvarc(tableid,varnum(tableid,'fname'),fname);
call putvarc(tableid,varnum(tableid,'ssn'),ssn);
if (append(tableid,'noinit')) then
        _msg_=sysmsg();
```

See Also

"OPEN" on page 576

"PUTVARC and PUTVARN" on page 609

"SET" on page 646

"UPDATE" on page 697

"INITROW" on page 480

APPLY

Invokes a method whose arguments are passed from an SCL list

Category: Object Oriented

Syntax

CALL APPLY(*control-id*,*method-name*,*arg-list-id*);

return-value=**APPLY**(*control-id*,*method-name*,
arg-list-id);

control-id
is the control whose method is being invoked.
Type: Numeric

method-name
is the method to invoke.
Type: Character

arg-list-id
contains the identifier of a list of arguments that the method requires. An invalid *arg-list-id* produces an error condition.
Type: Numeric

return-value
contains the value returned by *method-name*. The data type for *return-value* should match the data type for the called method.
Type: Character, List, Numeric, Object, Class, Interface.

Details APPLY provides the functionality of CALL SEND except that you can build a dynamic parameter list at run time instead of coding a fixed parameter list. You can also use APPLY as a function if the called method returns a value with a RETURN statement in the program that defines the method.

Example Instead of using the following statement to invoke a method that you have defined and named METHOD,

```
control.method(10,'abc','xyz',x);
```

you can use

```
args = makelist(4);
args = setitemn(args,10,1);
args = setitemc(args,'abc',2);
args = setitemc(args,'xyz',3);
args = setitemn(args,x,4);
call apply(control,'method',args);
```

More useful is the ability to combine APPLY with the ARGLIST= and REST= keywords in order to write methods that accept variable argument lists:

```
length _method_ $40;
m: method arglist=args;

call apply(otherControl,_method_, args);
```

This calls the method with the same arguments to the otherControl.
For example, a control receiving a method could rebroadcast the method to all controls on its _RECEIVERS_ list:

```
m: method arglist=args;
   _receivers_=getniteml(_self_,'_receivers_',
                         1, 1, 0);
   if _receivers_ then do
      r=listlen(_receivers_) to 1 by -1;
      call apply(getiteml(_receivers_, r),
                 _method_, args);
   end;
endmethod;
```

See Also

"NOTIFY" on page 572

"RETURN" on page 615

"SEND" on page 644

"SUPAPPLY" on page 679

"SUPER" on page 681

ARRAY

Defines elements of an explicit array

Category: Declarative Statement

Comparisons: SAS Statement with limitations in SCL

Syntax

ARRAY *array-name*<{*n*}><$><*length*><*elements*>
<(*initial-values*)>;

array-name
is the name of the array. It cannot be the same as the name of a window variable. However, window variables can be elements of an array.
Type: Character

{*n*}
is either the dimension of the array, or an asterisk (*) to indicate that the dimension is determined from the number of array elements or initial values. Multidimensional arrays are allowed. If an asterisk is specified without any array elements or initial values, then the array is a reference array. The dimension of this array will be determined at execution time, based on the corresponding array in the calling program.
Type: Numeric

$
indicates that the array type is character.

length
is the maximum length of elements in the array. For character arrays, the maximum length cannot exceed 200. The default length is 8 characters. *Length* is ignored for numeric arrays.
Type: Numeric

elements
are the variables (either window or nonwindow variables) that make up the array, or you can specify `'_TEMPORARY_'` to create a list of temporary data elements.
Type: Character

initial-values
are the values to use to initialize some or all of the array elements. Separate these values with commas or blanks. By default, all the elements in an array are initialized to missing.
Type: Character

Details If you have elements that you reference only with subscripting, then you can save memory by using the _TEMPORARY_ keyword. The SCL compiler has to allocate memory for the array name and the names of the array elements. However, if this keyword is used, the compiler allocates memory only for the array name. For large arrays, this could result in significant memory savings.

Note: Do not use **'_TEMPORARY_'** for *elements* if you plan to use the SET routine to fetch values from a SAS table directly into an array. Use GETVARN and GETVARC when **'_TEMPORARY_'** is specified. △

Note: You can also declare temporary arrays using the DECLARE statement. △

Reference Array A reference array is a pointer to another defined array. Previously, when an array needed to be passed as a parameter to a METHOD or ENTRY statement, an array of equal size needed to be defined in both the calling program and the called program. This technique used twice as much memory as was actually required. With reference arrays, only one array needs to be defined with the actual size. The array in the called program uses the actual memory of the array in the calling program.

By using reference arrays, you can create general array functions, because the array dimension is determined by the calling program. That is, you do not need to hardcode the array dimension in the SCL program that contains the ENTRY or METHOD statement. See the example later in this section for an illustration of this concept.

Using multidimensional reference arrays is allowed when the dimensions match. For example, if a two-dimensional array is passed in, the reference array must also be two-dimensional.

Reference arrays can currently be used as parameters only in a METHOD or ENTRY statement. Once a reference array has been created by a call to another program, it can be used in any way that a regular array can be used.

Differences from DATA Step in ARRAY Statement Execution The ARRAY statement in SCL is very similar to the ARRAY statement in the DATA step and is used to define single or multidimensional arrays. The ARRAY statement in SCL differs from the DATA step ARRAY statement in the following ways:

- SCL does not support implicitly subscripted arrays.
- SCL does not support the _NUMERIC_, _CHAR_, or _ALL_ keywords.
- SCL allows a repetition factor for initialization of arrays.
- SCL allows arrays to be used with the IN operator.
- SCL supports reference arrays.

For details about the ARRAY statement in the base SAS language, see *SAS Language Reference: Dictionary*.

Examples

Example 1: Using Repetition Factors for Array Initialization In the following statement, note that 1 is repeated three times and the pattern 2,3,4 is repeated four times:

```
array a{16}(0,3*1 ,4*(2,3,4));
```

This statement initializes the values of the elements of array A as follows:

```
0, 1, 1, 1, 2, 3, 4, 2, 3, 4, 2, 3, 4, 2, 3, 4
```

Example 2: Using an Array with the IN Operator Consider the following code segment:

```
array a 8 (2 4 6 8 10);

INIT:
   b=6;
   if b in a then put 'B is in array A';
      /* returns location of B in array A */
   c=b in a;
   put c=;
return;
```

This code produces the following output:

```
B is in array A
C=3
```

Example 3: Using a Reference Array with a METHOD Statement Assume that an entry SORT.SCL contains the method definition shown below. The method illustrates the use of a reference array to define a generic sort routine. The routine is termed generic because NSORT does not need to know the size of the array that is being passed: the reference array NARRAY takes on the definition of the array that is specified in the CALL METHOD routine.

```
nsort: method narray [*]:num;
   size = dim( narray );
   do i = 1 to size - 1;
      do j = i + 1 to size;
         if narray( i ) > narray( j ) then
            do;
               ntemp = narray( i );
               narray( i ) = narray( j );
               narray( j ) = ntemp;
            end;
      end;
   end;
endmethod;
```

Here is a sample calling program that executes the NSORT method:

```
array numarray(100);

MAIN:
  do i=1 to dim(numarray);
     numarray(i)=dim(numarray)-i+1;
  end;
  call method('sort.scl', 'nsort', numarray);
return;
```

See Also

"DECLARE" on page 330

ASORT

Sorts an array

Category: Array

Syntax

rc=**ASORT**(*array*<,*order*><,*elements*>);

rc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

array
is an SCL array that was declared in an ARRAY statement.
Type: Character

order
specifies the order for the sort:

'A'	ascending order (the default)
'D'	descending order

Type: Character

elements
is the number of elements to sort.
Type: Numeric

Details By default, the array is sorted in ascending order. You can use the optional *order* argument to specify either ascending or descending order.

By default, the entire array is sorted. You can use the optional *elements* argument to restrict sorting to the specified number of elements (starting from the beginning of the array).

If the value of the *elements* argument is greater than the total number of array elements, the program halts execution and sends an error message to the log.

Example

Sort the first five items of array VALUES in ascending order:

```
array values[8] val1-val8 (3,5,2,1,6,8,7,4);

if (asort(values,'d',5)) then _msg_=sysmsg();
else do;
   _msg_='Sort was successful';
   do i=1 to dim(values);
      put values{i}=;
   end;
end;
```

This produces the following output:

```
values[ 1 ]=6
values[ 2 ]=5
values[ 3 ]=3
values[ 4 ]=2
values[ 5 ]=1
```

```
values[ 6 ]=8
values[ 7 ]=7
values[ 8 ]=4
```

ATTRC and ATTRN

Return the value of an attribute for a SAS table

Category: SAS Table

Syntax

attr-value=**ATTRC**(*table-id*,*attr-name*);

attr-value=**ATTRN**(*table-id*,*attr-name*);

attr-value
contains the value of the SAS table attribute. For ATTRC, the value is for a character attribute. For ATTRN, the value is for a numeric attribute.
Type: Character, Numeric

table-id
contains the identifier for the SAS table. This is the identifier that was returned by the OPEN function when the table was opened. If *table-id* is invalid, the program halts.
Type: Numeric

attr-name
is the name of the SAS table attribute. If *attr-name* is invalid, then a missing or null value is returned for *attr-value*. Values that can be used with ATTRC are listed in "Attributes for the ATTRC Function" on page 258. Values that can be used with ATTRN are listed in "Attributes for the ATTRN Function" on page 260.
Type: Character

Attributes for the ATTRC Function

To check for a character attribute of a SAS table, use ATTRC with one of the following values for *attr-name*:

'CHARSET'
returns a string indicating the character set of the machine that created the SAS table. It returns either one of the following values, or an empty string if the SAS table is not sorted:

ASCII
ASCII character set

EBCDIC
EBCDIC character set

HASCII
extended ASCII character set

ANSI
OS/2 ANSI standard ASCII character set

OEM
: OS/2 OEM code format

'ENCRYPT'
returns **'YES'** or **'NO'** depending on whether the SAS table is encrypted.

'ENGINE'
returns the name of the engine used to access the SAS table.

'LABEL'
returns the label assigned to the SAS table.

'LIB'
returns the libref of the SAS data library in which the SAS table resides.

'MEM'
returns the name of the SAS data library member.

'MODE'
returns the mode in which the SAS table was opened, such as:

I	INPUT mode, allows random access if the engine supports it; otherwise, defaults to IN mode
IN	INPUT mode, reads sequentially and allows revisiting rows
IS	INPUT mode, reads sequentially but does not allow revisiting rows
N	NEW mode (to create a new SAS table)
U	UPDATE mode, allows random access if the engine supports it; otherwise, defaults to UN mode
UN	UPDATE mode, reads sequentially and allows revisiting rows
US	UPDATE mode, reads sequentially but does not allow revisiting rows
V	UTILITY mode, allows modification of column attributes and indexes that are associated with the SAS table.

For more information about open modes, see “OPEN” on page 576.

'MTYPE'
returns the type of the SAS data library member.

'SORTEDBY'
returns an empty string if the SAS table is not sorted. Otherwise, returns the names of the BY columns in the standard BY statement format.

'SORTLVL'
returns an empty string if the SAS table is not sorted. Otherwise, returns one of the following:

WEAK
: The sort order of the SAS table is not validated. That is, the sort order was established by a user (for example, through a SORTEDBY SAS data set option). The system cannot validate its correctness, so the order of the rows cannot be depended upon.

STRONG
: The sort order of the SAS table is validated. That is, the order of its rows can be depended upon. The sort order was established by the software (for example, through PROC SORT or through the OUT= option on the CONTENTS procedure).

'SORTSEQ'
returns an empty string if the SAS table is sorted on the native machine or if the sort collating sequence is the default for the operating system. Otherwise, returns the name of the alternate collating sequence that is used to sort the file.

'TYPE'
is the SAS table type.

Attributes for the ATTRN Function

To check for a numeric attribute, use ATTRN with one of the following values for *attr-name*:

'ANY'
specifies whether the table has rows or columns:

-1	The table has no rows or columns.
0	The table has no rows.
1	The table has both rows and columns.

'ALTERPW'
indicates whether a password is required in order to alter the SAS table:

1	The SAS table is alter protected.
0	The SAS table is not alter protected.

'ANOBS'
indicates whether the engine knows the number of rows:

1	The engine knows the correct number of rows.
0	The engine does not know the correct number of rows.

'ARAND'
indicates whether the engine supports random access:

1	The engine supports random access.
0	The engine does not support random access.

'ARWU'
indicates 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.

'CRDTE'
returns the SAS table creation date. The value returned is the internal SAS DATETIME value for the creation date. Use the DATETIME format to display this value.

'GENMAX'
returns the maximum number of generations.

'GENNEXT'
returns the next generation number to generate.

'ICONST'
returns information on the existence of integrity constraints for a SAS table:

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'
indicates whether the SAS table supports indexing:

1	Indexing is supported.
0	Indexing is not supported.

'ISINDEX'
indicates whether the SAS table is indexed:

1	At least one index exists for the SAS table.
0	The SAS table is not indexed.

'ISSUBSET'
indicates whether the SAS table is a subset:

1	At least one WHERE clause is active.
0	No WHERE clause is active.

'LRECL'
returns the logical record length.

'LRID'
returns the length of the record ID.

'MODTE'
returns the last date and time the SAS table was modified. Use the DATETIME format to display this value.

'NDEL'
returns the number of deleted rows in the SAS table.

'NLOBS'
returns the number of logical rows (those not marked for deletion). An active WHERE clause does not affect this number.

'NLOBSF'
returns the number of logical rows (those not marked for deletion) that match the active WHERE clause.

Note: NLOBSF should be used with caution. Passing NLOBSF to ATTRN requires the engine to read every row from the table that matches the WHERE clause. Based on the file type and size, this can be a time-consuming process. △

'NOBS'
returns the number of physical rows (including those marked for deletion). An active WHERE clause does not affect this number.

'NVARS'
returns the number of columns in the SAS table.

'PW'
indicates whether a password is required in order to access the SAS table:

1	The SAS table is protected.
0	The SAS table is not protected.

'RADIX'
indicates whether access by row number is allowed:

1	Access by row number is allowed.

0	Access by row number is not allowed.

Note: A SAS table on a tape engine is index addressable even though it cannot be accessed by row number.

'READPW'
indicates whether a password is required in order to read the SAS table:

1	The SAS table is read protected.
0	The SAS table is not read protected.

'TAPE'
indicates whether the SAS table is a sequential tape file:

1	The SAS table is a sequential tape file.
0	The SAS table is not a sequential tape file.

'WHSTMT'
returns information about 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'
indicates whether a password is required in order to write to the SAS table:

1	The SAS table is write protected.
0	The SAS table is not write protected.

Examples

Example 1: Using the ATTRC Function Ensure that the SAS table has been opened in UPDATE mode and display an error message if it is not open:

```
mode=attrc(tableid,'MODE');
if (mode ne 'U') then _msg_=
   'Table not open in UPDATE mode.';
else rc=sort(tableid,'name');
```

Example 2: Using the ATTRN Function Determine whether a WHERE clause is currently active for a SAS table:

```
iswhere=attrn(tableid,'whstmt');
if (iswhere) then _msg_=
   'A WHERE clause is currently active.';
```

See Also

"DESCRIBE" on page 341

"OPEN" on page 576

BLANK

Special missing value

Category: System Variable

Details _BLANK_ is a system variable that is created for every SCL program that you compile. The compiler creates a space for _BLANK_ in the SCL data vector. In SAS/AF applications, you can compare the value of window variables against the value _BLANK_ to test whether a value has been entered in a field in the window. The test is valid for both numeric and character variables. You can also use _BLANK_ in assignment statements to reset a window variable to a blank, as if the user had not entered a value in the field. You cannot reset the value of the _BLANK_ variable itself.

In comparison operations, _BLANK_ is considered the smallest missing value.

Example

The following code fragment prints a message if X is modified and is blank:

```
if modified(x) and x eq _blank_ then
   _msg_ = 'Please enter a value';
```

See Also

"CLRFLD" on page 293

"SETFLD" on page 650

BLOCK

Displays a menu containing up to 12 choice blocks and returns the number of the user's choice

Category: Window

Syntax

choice=**BLOCK**(*window-name*,*title*,*color*,*text-1*, . . . ,
text-12<,*icon-1*, . . . , *icon-12*>);

choice
returns either the number (1-12) of the selected block, or one of the following:

–99
if a user requested help for SAS software

–1 to –12
if a user requested help for the block

0
if a user issued the END, CANCEL, or BYE command

99
if an unknown command is issued (see "WORD" on page 725).

Type: Numeric

window-name
is the title (up to 80 characters) for the window.
Type: Character

title
is the title (up to 60 characters) for the menu's title box.
Type: Character

color
is a number from 0 to 33 that represents the combination of colors to be used for the blocks. The colors and numbers are listed in Table 16.1 on page 265. Some devices do not support changing the background color. If you have specified that icons be used with BLOCK, then the specified color combination may not take effect. The display of icons is host specific, and therefore the color may be controlled by the host operating system. (Under Windows or OS/2, use the Color Palette to alter icon colors. Under X windows, set X resources to control icon colors.)
Type: Numeric

text-1, . . . , text-12
is the text for each block or icon to display (up to 14 characters). The blocks or icons are displayed in groups of four. Blocks for the first four *text* values are displayed on the first row, blocks for the second four *text* values are displayed on the middle row, and blocks for the last four *text* values are displayed on the last row. Twelve values are required, but you can use null values for block positions that you do not want displayed.
Type: Character

icon-1, . . . , icon-12
are numbers for icons to display in place of the blocks. If no values are provided for icons, or if the host system does not support icons, then standard rectangular blocks are displayed. If you specify a number for which no icon is defined, then the default SAS icon is displayed. If at least one icon number is specified, and the total number of icons is less than the total number of text labels, then the default SAS icon is displayed for text labels that lack an associated icon number.
Type: Numeric

Details The number of text values that you specify determines how many blocks of icons are displayed in the menu. In order to display an icon menu, you must specify at least one icon position, although you can display the default SAS icon by specifying 0 for positions for which a value for *text* is supplied.

Because BLOCK does not generate a physical display window, window options such as KEYS are not recognized. The BLOCK function windows recognize only DMKEYS settings. To alter DMKEYS settings for a BLOCK menu, you can use GETFKEY and SETFKEY in a program that runs before the BLOCK menu opens. This program must have a display screen associated with it.

This function attempts to display the blocks in the best manner depending upon the characteristics of the user's display device. The function displays up to three rows of four blocks. Blocks are displayed in the order in which the *text* arguments appear in the function. Only the nonblank choices are displayed, and the blocks in each row are centered in the row.

When the function is called, it queries the current value of *choice*. If the value of *choice* corresponds to a valid selection number, the cursor is positioned on the correct block. Otherwise, the cursor is positioned in the upper-left corner of the window.

To make a selection from the block menu, a user must move the cursor to the appropriate block and then press ENTER or click the mouse. BLOCK returns the index of the selection.

If a user presses the HELP key on one of the selections, then the negative of the selection is returned. If a user presses the HELP key while the cursor is not on one of the blocks, then the value –99 is returned.

If a user issues the END or CANCEL command via a function key, then the value 0 is returned.

Use ENDBLOCK to close the menu window that is opened by BLOCK. Table 16.1 on page 265 lists values that can be used for *color*.

Values For The Color Argument

Table 16.1 Values for the *Color* Argument

Color	Background	Border	Text	Icon	Shadow
0	black	white	white	white	white
1	black	gray	cyan	gray	blue
2	black	gray	cyan	gray	blue
3	black	cyan	cyan	gray	cyan
4	black	gray	white	cyan	gray
5	black	cyan	yellow	cyan	blue
6	black	gray	white	blue	gray
7	black	gray	yellow	blue	gray
8	black	gray	white	red	gray
9	black	gray	white	pink	gray
10	black	gray	white	yellow	gray
11	black	gray	white	red	blue
12	blue	gray	cyan	gray	black
13	blue	gray	yellow	gray	black
14	blue	gray	white	gray	black
15	blue	gray	white	magenta	black
16	blue	gray	white	red	black
17	blue	gray	white	cyan	black
18	blue	yellow	white	yellow	black
19	blue	gray	white	magenta	gray
20	blue	gray	white	red	gray
21	gray	blue	black	blue	black
22	gray	red	black	red	black
23	gray	magenta	black	magenta	black
24	gray	blue	black	cyan	blue
25	gray	cyan	black	cyan	black
26	red	gray	white	gray	black
27	red	gray	black	gray	black
28	pink	gray	white	gray	black
29	pink	gray	black	gray	black
30	yellow	gray	black	gray	black
31	brown	gray	gray	gray	black

Color	Background	Border	Text	Icon	Shadow
32	background*	border*	foreground*	secondary background*	black
33	secondary background*	secondary border*	foreground*	background*	black

* SASCOLOR window element names.

Example

Create a menu with five choices represented by icons. The first row contains two icons, **`Outline`** and **`Index`**. The second row contains two icons, **`Compare Files`** and **`Calendar`**. The third row contains a single icon, **`End`**.

Because the CHOICE variable is assigned the value 5 before BLOCK is called, the cursor is positioned on the Compare Files icon when the window opens. When a user makes a selection, the SELECT statement either exits the DO loop or calls another PROGRAM entry. When control returns from the called program, the menu is displayed again.

```
INIT:
choice=5;
LOOP:
   do while(choice ne 0);
      choice=block('Writers Toolbox',
        'Main Menu',6,'Outline','Index',
         '','','Compare Files',
         'Calendar','','',
         'End','','','',
         1,2,0,0,3,4,0,0,111,0,0,0);
      select(choice);
         when(1) call display('outl.scl');
         when(2) call display('index.scl');
         when(5) call display('compare.scl');
         when(6) call display('calend.scl');
         when(9) leave LOOP;
         otherwise do;
            if (choice<0) then
            call display('help.scl',choice);
         end;
      end;
   end;
   call endblock();
return;
```

See Also

"ENDBLOCK" on page 364

BUILD

Invokes the BUILD window in SAS/AF software

Category: Utility

Syntax

CALL BUILD(*entry*<,*open-mode*<,*resource*><,*pmenu*>>);

entry
is the name of an entry in a SAS catalog. A one-level name is assumed to be WORK.*catalog*. A two-level name is assumed to be *libref.catalog*. A three-level name is assumed to be *libref.catalog.entry*.PROGRAM. A four-level name is assumed to be *libref.catalog.entry.type*.

If a catalog is specified instead of a catalog entry, the Explorer window is brought up and the three optional parameters are ignored.

Type: Character

open-mode
is the mode in which to open the catalog:

'EDIT'|'E'
opens the catalog entry for editing. (This is the default.)

'BROWSE'|'B'
opens the catalog entry for browsing.

'COMPILE <NOMSG>'|'C <NOMSG>'
compiles the FRAME, PROGRAM, or SCL entry specified in *entry*. The **NOMSG** option prevents NOTE messages from being sent to the SAS log when batch files are being run (or to the LOG window for all other files) when the program is compiled, but it does not suppress compiler error or warning messages.

Type: Character

resource
is the RESOURCE entry if *entry* is a FRAME entry. A one- or three-level name can be specified. A one-level name assumes that the RESOURCE entry is in the current catalog, the SASUSER.PROFILE catalog, or the SASHELP.FSP catalog.

Type: Character

pmenu
is the PMENU entry for the DISPLAY window. If *pmenu* is not supplied, a default PMENU entry is displayed. A one-level name assumes that the PMENU entry is in either the current catalog, the SASUSER.PROFILE catalog, or the SASHELP.FSP catalog. For FRAME and PROGRAM entries, you can specify a secondary PMENU entry for the SOURCE window. Use a space to separate a secondary PMENU name from a main PMENU name.

Type: Character

Details You can use the BUILD routine to

- open catalog entries from within an application
- compile PROGRAM, FRAME, or SCL entries without displaying the contents of the entries
- view SAS/GRAPH or SAS/AF entries.

If a catalog entry is specified in the first parameter instead of an entry, none of the optional parameters are honored.

Note: When a program uses the BUILD routine, SAS/AF software must be licensed at sites where the application is executed. △

Examples

Example 1: Opening an Entry Open a PROGRAM entry named NAME in edit mode in the catalog MYLIB.MYCAT:

```
call build('mylib.mycat.name');
```

Example 2: Compiling a FRAME Entry Compile the source code for the entry A.FRAME in the WORK.TEST catalog without opening a DISPLAY window for the entry and without displaying any NOTES:

```
call build('work.test.a.frame','compile nomsg');
```

Example 3: Specifying RESOURCE and PMENU Entries Edit a FRAME using a particular RESOURCE and PMENU:

```
call build('lib.cat.name.frame','e',
   'lib.cat.build.resource','lib.cat.build.pmenu');
```

Example 4: Specifying a Secondary PMENU Entry Edit a FRAME entry using a custom PMENU for both the DISPLAY and SOURCE windows. The DISPLAY window uses MYPMENU1.PMENU, and the SOURCE window uses MYPMENU2.PMENU.

```
call build('lib.cat.a.frame','e','',
     'mypmenu1 mypmenu2');
```

See Also

"CBT" on page 274

"DISPLAY" on page 350

CATCH

Processes an exception that has been thrown with the THROW statement

Category: Control Flow

Syntax

CATCH *exception*;
    / SCL statements to process the exception */*

ENDCATCH;

Note: CATCH blocks must always be enclosed in DO statements. △

exception
is the local variable for the exception (which is an instance of the SCLException class) that you want to process.

Details

When an exception is raised via the THROW statement, normal execution of the program stops, and SCL begins looking for a CATCH block to process the exception.

The CATCH block can contain any statements needed to process the exception, including additional CATCH and THROW statements.

SCL uses the scope of the DO group that contains the CATCH block and the class of the exception to determine which CATCH block to execute. For details, see "How SCL Determines Which CATCH Block To Execute" on page 160.

Each entry in the stack can process an exception and then pass it back up the stack by rethrowing it, which allows the calling entry to perform additional processing. Each entry can perform whatever processing is relevant to that entry.

If an exception is rethrown within a CATCH block, no other CATCH block within the same scope can recatch the exception. The exception is passed out of the scope where it was thrown. Also, you cannot define multiple CATCH blocks for the same exception within the same scope.

Example

The following DO group declares a local exception variable called NE, creates a new instance of NE, and throws the new exception. The CATCH block prints the traceback information that is automatically stored by SCL when an exception is thrown.

```
do;
dcl SCLException NE = _new_ NewException('Exception in method m');
throw NE;

catch NE;
   put NE.getMessage();         /* Print exception information. */
   call putlist(NE.traceback);
endcatch;
end;
```

See Also

"THROW" on page 688

Chapter 10, "Handling Exceptions," on page 157

CATLIST

Displays a host selector window that lists entries in a SAS catalog, and returns user selections

Category: Catalog, Selection List

Syntax

selections=**CATLIST**(*catalog-name*,*type*,*num-sel*,*prefix*<,*message*>);

selections

contains one or more user selections from the list. Separate multiple selections with one or more blanks. By default, *selections* is 200 bytes long. To accommodate values longer than 200 bytes, explicitly declare *selections* with a longer length.

Type: Character

catalog-name
is either a SAS catalog name, in the form *libref.catalog*, or `*` to allow a user to interactively select a libref, catalog, and entry.
Type: Character

type
is the entry type to list for selection (for example, SCL or FRAME). To display the names of all entries in the catalog, use **`'ALL'`** or `''`.

num-sel
is the maximum number of items a user can select. To display the list for information only (no selections allowed), specify **`0`**. To specify an unlimited number of selections, use a value that is equal to or larger than the number of available selections. A user cannot make a number of selections that exceeds the number of items in the list.
Type: Numeric

prefix
specifies whether selected entries are prefixed by the catalog name:

`'Y'`	returns selected names in the form *libref.catalog.entry.type*. This is the default value if *catalog-name* is *.
`'N'` | `''`	returns selected names in the form *entry.type*.

Type: Character

message
is the text for the message that is displayed at the top of the selection list window. The default message tells users to make up to *num-sel* selections.
Type: Character

autoclose
is an obsolete argument but is retained for compatibility with earlier releases. If you want to specify a value for *num-sel*, then specify `''`as a placeholder for this argument.
Type: Character

Details You can provide default values that will be initially selected when the catalog selection list is displayed. To do this, assign the values to the *selections* variable before calling CATLIST. If *selections* contains valid values when the function is invoked, then those names are automatically designated as selected when the selection list is displayed.

If a user closes the Catalog Entry Selector window without making a selection, then CATLIST returns a blank value unless there was an initial value for the *selections* variable before CATLIST was called.

Selections from the window can be returned in the current result list, if one is available. The current result list is a special SCL list that is automatically filled with the values that have been selected from a selection list. To create a current result list, use the MAKELIST function to create it, and use the CURLIST function to designate it as the current result list. The current result list must exist before you call the CATLIST function.

When CATLIST is invoked, the current result list is cleared. After CATLIST is invoked, the result list contains the following named items:

TAG
identifies the list as one that was created by CATLIST.
Type: Character

COUNT
contains either the number of selected elements, or 0 if a user makes no selections or issues a CANCEL command in the list window.
Type: Numeric

NAME
: contains the uppercase name of each selected catalog entry. If the value of *prefix* is **Y**, then the prefix is appended to the beginning of each name. There is one NAME element for each selection.
  Type: Character

DESC
: contains the description of each selected catalog entry. There is one DESC element for each selection. The value of DESC is in the same case that was entered originally.
  Type: Character

DATE
: contains the date of last modification for each selected catalog entry. There is one DATE element for each selected catalog entry.
  Type: Character

Because some engines support mixed-case filenames, CATLIST now retains the cases of the returned selected items. This may cause your application to fail if your application contains code that assumes the returned selection is uppercased. For example,

```
if (catlist(dsid, 'TESTNDX')='NDXVAR')
```

must be changed to

```
if (upcase(catlist(dsid, 'TESTNDX'))='NDXVAR'
```

If the application cannot be modified, you may need to specify the VALIDVARNAME=V6 system option when you run the application to ensure that the selections returned from the CATLIST function will be uppercased.

Example

Display a selection list that contains the entries in the catalog MYLIB.TEST, and allow users to make up to five selections. Use GETNITEMC to retrieve the selected values from the current result list.

```
listid=makelist();
rc=curlist(listid);
selections=catlist('mylib.test','all',5);
n=getnitemn(listid,'COUNT');
do i=1 to n;
   name=getnitemc(listid,'NAME',i);
   desc=getnitemc(listid,'DESC',i);
   date=getnitemc(listid,'DATE',i);
   put name= desc= date=;
end;
```

See Also

"CURLIST" on page 320

"DIRLIST" on page 347

"FILELIST" on page 401

"LIBLIST" on page 513

CATNAME

Defines a concatenated catalog, which contains a logical combination of the entries in two or more catalogs

Category: Catalog

Syntax

rc=**CATNAME**(*cat-name*,*action*<,*catalog-list*><,*list-id*>);

rc
contains the return code for the operation:

0	successful
1	not successful

Type: Numeric

cat-name
is a two-level name (*libref.catalog*) for the concatenated catalog. The libref must already be defined.
Type: Character

action
specifies the action to take on *cat-name*:

'SET'	sets the definition.
'CLEAR'	clears the definition.
'LIST'	lists the members and saves them in the SCL list referenced by *list-id*.

Type: Character

catalog-list
lists the two-level names (*libref.catalog*) of two or more SAS catalogs whose entries are logically combined in *cat-name*. Use at least one blank space to separate catalog names. This list can contain concatenated catalogs that were previously defined with the CATNAME function.
Type: Character

list-id
contains the identifier for the list of catalogs to be logically combined in *cat-name*, if *action* is **SET** and *catalog-list* is not specified. If *action* is **LIST**, then the list contains the following information about all of the entries:

CATNAME
The catalog name, which is a two-level name (libref.catalog) for the concatenated catalog.
Type: Character

LEVELS
The total number of levels of the defined catalog name.
Type: Numeric

(sublist1...sublist*n*)
Sublist number 1 to *n*, where *n* is the total number of levels.
Type: List
The sublist elements are as follows:

- **LEVEL**, which is the level number. Type: Numeric
- **CATALOG**, which is the catalog name. Type: Character
- **ENGINE**, which is the engine name. Type: Character
- **PHYSICAL NAME OF LIBRARY**, which is the physical name of the library. Type: Character

Type: Numeric or List

Details CATNAME defines a concatenated catalog that is a logical combination of the entries in two or more SAS catalogs. When the program uses *cat-name* as the catalog in a catalog entry name, SAS searches for the specified entry in the catalogs specified in *catalog-list* or *list-id*. A concatenated catalog name that is defined with CATNAME can be part of *catalog-list* or *list-id* for another CATNAME function.

This feature is most useful for debugging and testing during application development. It eliminates the need to copy a whole catalog when you need to test and change a single catalog entry that contains a particular method.

Examples

Example 1: Defining a CATNAME Create concatenated catalog S.T, which logically combines the entries in catalogs A.B and C.D. Then, create concatenated catalog E.F, which contains the logical combination of the entries in the concatenated catalog S.T as well as the entries in Q.T and X.Y. A subsequent reference in a program to S.T.TEST.SCL causes SAS to search for A.B.TEST.SCL and then for C.D.TEST.SCL. A reference to E.F.TEST.SCL causes SCL to search for A.B.TEST.SCL, then for C.D.TEST.SCL, then for Q.T.TEST.SCL, and finally for X.Y.TEST.SCL.

```
rc=catname ('s.t','set','a.b c.d');
rc=catname ('e.f','set','s.t q.t x.y');
```

Example 2: Defining a CATNAME for a *list-id* Use the 'LIST' action to query the definition of E.F from Example 1.

```
list=makelist();
rc=catname('e.f','list','',list);
call putlist(list);
```

See Also

"SEARCH" on page 635

CBT

Runs a CBT entry

Category: Modular Programming and Object Oriented

Syntax

CALL CBT(*entry*<,*frame*><,*frame-name*>);

entry
is the name of a CBT entry. Specify a CBT entry in the current catalog with *entry*. Specify an entry in the current catalog with *entry.type*. Specify a CBT entry in a different catalog with *libref.catalog.entry*. Specify a different type of entry in a different catalog with *libref.cat-name.entry.type*.

frame
is the number of the frame in the CBT entry to be displayed initially.
Type: Numeric

frame-name
is the name assigned to the frame in the CBT to be displayed initially. If this argument is specified, the value of *frame* is ignored.
Type: Character

Details The CBT routine opens a CBT entry window. You can optionally specify the name or number of the CBT frame to display initially. For information about CBT entries, see the SAS documentation for SAS/AF software. You can use CBT entries to link detailed instructional information with an application so that the information is readily available.

The following restrictions apply for users of a CBT entry that is called from an SCL program:

- The SAVE command is not recognized.
- =X returns the user to the calling program.
- QCANCEL returns the user to the calling program.
- QEND returns the user to the calling program.

Examples

Example 1: Displaying a Particular CBT Frame Display the second frame of the entry EXAMPLE.CBT:

```
call cbt('example',2);
```

Display the frame named ABC of the entry EXAMPLE.CBT:

```
call cbt('example',1,'abc');
```

Example 2: Calling a CBT Entry Suppose an application requires users to be familiar with a particular set of terminology. You use the following program to call a CBT course (in this example, TERMINAL.CBT) that teaches the appropriate information. The following example runs the CBT entry if a user issues the command TEACH. Then, when the user exits the CBT course, it returns control to the SCL statement that follows the call to TERMINAL.CBT.

```
control always;
if (upcase(word(1))='TEACH') then
   do;
      call nextcmd();
      call cbt('terminal');
   end;
```

See Also

"DISPLAY" on page 350

"GOTO" on page 455

CENTER

Returns a centered character string

Category: Character

Syntax

centered-string=**CENTER**(*string*<,*length*>);

centered-string
contains the centered character string. If *centered-string* already exists, then specifying *length* changes the length of *centered-string* to the number of characters specified in *length*.
Type: Character

string
contains the character string to be centered.
Type: Character

length
contains the length in which the character string is to be centered. The default is the maximum length of *centered-string*.
Type: Numeric

Details The default length is the maximum length of *centered-string*. The string is centered by padding with spaces. To enable CENTER to work properly when *centered-string* is a window variable, set the justification attribute (**JUST**) for the control or field to **NONE**.

To left- or right-justify a string, use LEFT or RIGHT, respectively.

See Also

"LEFT" on page 506

"RIGHT" on page 620

CEXIST

Verifies the existence of a SAS catalog or SAS catalog entry

Category: Catalog

Syntax

rc=**CEXIST**(*entry*< ,'U'>);

rc
contains the return code for the operation:

1	The SAS catalog or catalog entry exists.
0	The SAS catalog or catalog entry does not exist.

Type: Numeric

entry
is SAS catalog, or the name of an entry in a catalog. A one- or two-level name is assumed to be the name of a catalog. To test for the existence of an entry within a catalog, use a three- or four-level name.

Type: Character

'U'
tests whether the catalog specified in *entry* can be opened for updating.

Type: Character

Examples

Example 1: Testing Whether a Catalog Can Be Updated Test whether the catalog LIB.CAT1 exists and can be opened for update. If the catalog does not exist, a message is displayed on the message line.

```
if (cexist('lib.cat1','u')) then
   _msg_='The catalog LIB.CAT1 exists and can be
                            opened for update.';
else _msg_=sysmsg();
```

Example 2: Verifying the Existence of a Catalog Verify the existence of the entry X.PROGRAM in LIB.CAT1:

```
if (cexist('lib.cat1.x.program')) then
   _msg_='Entry X.PROGRAM exists';
else _msg_=sysmsg();
```

See Also

"EXIST" on page 383

CFRAME

Contains the identifier of the FRAME entry that is currently executing

Category: System Variable

Details _CFRAME_ is a system variable that is provided automatically for FRAME entries. _CFRAME_ has a valid value only when a FRAME entry's SCL code is running or when a FRAME entry or component method is running.

Example

Suppose the entry FIRST.FRAME contains an icon. The icon's _select method is defined to run SECOND.FRAME, which contains the following program:

```
INIT:
       /* Send a method to the current FRAME */
   _CFRAME_._setMsg('Running the Select method');
return;
TERM:
       /* Send a method to the FRAME that  */
       /* contains the icon                */
   _FRAME_._setMsg('The Select method has finished.');
return;
```

When FIRST.FRAME displays and a user selects the icon, SECOND.FRAME displays with the message "Running the Select method." After the user ends from SECOND.FRAME, FIRST.FRAME displays the message "The Select method has finished." This is accomplished by sending the _setMsg method to _CFRAME_ (the FRAME entry that is currently running) in the INIT section and by sending _setMsg to _FRAME_ (the FRAME entry that contains the icon) in the TERM section.

See Also

"_FRAME_" on page 429

CLASS

Creates a class using SCL code

Category: Object Oriented

Syntax

<**ABSTRACT**> **CLASS** *class-name*<EXTENDS *parent-class-name*>
    <SUPPORTS *supports-interface-clause*>

<REQUIRES *requires-interface-clause*>
< / (*class-optional-clause*)>
<(*attribute-statements*)>
<(*method-declaration-statements*)>
<(*method-implementation-blocks*)>
<(*event-declaration-statements*)>
<(*eventhandler-declaration-statements*)>
ENDCLASS;

ABSTRACT
is an optional keyword used to define a class as an abstract class. Methods defined inside an abstract class are not required to have method implementations. Abstract classes are used to specify a common interface for several classes. An abstract class can not be instantiated through the _NEW_ operator.

class-name
is the name of a class that you are creating, which can be specified as a one- to four-level name.

parent-class-name
specifies the parent class of *class-name* and is specified as EXTENDS *parent-class-name*. *Parent-class-name* can be specified as a one- to four-level name.

If no EXTENDS clause is supplied, *parent-class-name* defaults to SASHELP.FSP.OBJECT.CLASS, which is the parent class of all SAS/AF classes.

supports-interface-clause
lists the interfaces that this class supports. Interfaces can be specified as follows:

SUPPORTS*interface-1*<,*interface-2*...>

The *interface* names can be specified as one- to four-level names. All of the methods listed in SUPPORTS must be implemented in the CLASS block.

requires-interface-clause
lists the interfaces required for this class. Interfaces are specified as follows:

REQUIRES *interface-1*<,*interface-2*...>

The *interface* names can be specified as one- to four-level names. The REQUIRES interfaces are used for the client-server model. For more information, see "INTERFACE" on page 486.

class-optional-clause
specifies options for the class. Options should be placed inside parentheses following a / (slash). Separate multiple options with commas. Class options can be any of the following:

Description=*description*
is a description of the CLASS entry

MetaClass=*class-name*
is the four-level name of the CLASS entry that contains the model of a class. The default MetaClass is SASHELP.FSP.CLASS.CLASS.

attribute-statements
defines the class attributes, which determine how an object will look and behave. Attributes can either be scalars or arrays. The syntax of a class attribute statement is: *access-scope type var-name*< / (*attribute options*)>;

access-scope
can be one of the following:

'PUBLIC'
: specifies that the attribute can be accessed by any SCL program. DECLARE may be used in place of public scope.

'PRIVATE'
: specifies that the attribute can be accessed only from methods in the class where the attribute is defined.

'PROTECTED'
: specifies that the attribute can be accessed only from methods in subclasses of the class where the attribute is defined. Since a class can be considered a subclass of itself, a protected attribute can also be accessed from the class where it is defined.

type
: is the data type of the attribute. NUM, CHAR, LIST, OBJECT or a four-level *class-name* are possible values of *type*.

var-name
: is the name of the attribute. You can specify a multi-dimensional array by providing an array dimension after *var-name*. For example:

```
PRIVATE num narray{3, 2, 3};
```

If an array variable has the same name as a method, the method name has higher precedence when referencing that array. To avoid ambiguity, use **[ ]** or **{ }** instead of **( )** to specify the array reference.

attribute-options
: specifies options for a class attribute. List options inside parentheses following a / (slash). Separate multiple options with commas. *Attribute-options* can be any of the following:

AutoCreate='**N**' | '**Y**'
: determines whether an SCL list is created automatically when an instance of the class is created. If **AutoCreate='Y'** (default), a four-level object name or SCL list is created depending on the attribute type. If **'N'**, then a four-level object name or SCL list is not created, and the user is responsible for creating and deleting this list.

Category=*category-name*
: specifies the category for an attribute. Categories organize attributes so that you can display only attributes for the category. You can create your own category names. Components that are supplied with SAS software belong to the following categories:

Appearance

Data

Drag and drop

Help

Misc (Miscellaneous)

Region

Size/location

Misc is the default.

Description=*attribute-description*
: specifies the description of the attribute. When you click on an attribute in the Class Editor, this text is displayed below the list of attributes.

Editable=**'N'** | **'Y'**
determines whether attributes can be altered. **'Y'** is the default.

If **EDITABLE='Y'**, then the attribute can be set anywhere that it is in scope:

- □ If the attribute is defined in class C and it is a public attribute, then it can be set anywhere.
- □ If the attribute is defined in class C and it is a private attribute, then it can only be set from methods in the class C.
- □ If the attribute is defined in class C and it is a protected attribute, then it can only be set from methods in C and subclasses of C.

If **EDITABLE='N'**, then the ability to set the attribute is restricted based on its scope:

- □ If the attribute is defined in a class C and it is a public attribute, then it can only be set from methods in C and subclasses of C.
- □ If the attribute is defined in class C and it is a protected attribute, then it can only be set from methods in C.
- □ If the attribute is defined in class C and it is a private attribute, it cannot be set anywhere. (It is effectively a constant.)

Editor=*editor-entry-name*
specifies a FRAME, SCL, or PROGRAM entry that returns a value. The **Editor**= option is available for attributes of all types except OBJECT. If supplied, the specified entry is displayed and executed by the Properties window when the ellipsis button (...) in the cell is clicked. The value that is returned from the entry is displayed in the cell in the Properties window.

Editors are typically FRAME entries that are designed to aid an application developer in specifying a value for an attribute. For example, for an attribute called 'textColor' that can be assigned to any hexcolor string, you could design a FRAME entry window to help the user visually see what the hexcolor string represents. The window could contain an RGB slider control with a sample box that shows the color that is being created as a user manipulates the red/green/ blue sliders. In this example, you assign the name of the FRAME entry as the value of **EDITOR**=, and this window opens when a user selects the **...** button for the TEXTCOLOR attribute in the Properties window.

GetCAM=*method-name*
specifies the custom access method to be executed when the value of the attribute is queried. Using dot notation to query an attribute for which a getCAM method has been defined may result in side effects. See "What Happens When Attribute Values Are Set or Queried" on page 122.

InitialValue=*initial-values*
specifies an initial value for the attribute. This option is valid only for attributes with types CHAR , NUM, and SCL LIST.For an example of using an SCL list as an initial value, see "Initializing the Values in a List" on page 51.

Linkable=**'N'** | **'Y'**
determines whether an attribute is linkable from the Properties window. Only public attributes are linkable. Private and protected attributes are not displayed in the Properties window. **Y** is the default.

SendEvent=**'N'** | **'Y'**
determines whether an attribute sends an event when modified. When **SENDEVENT='Y'**, SAS assigns the Event Name, which has the format **''***attributeName* **Changed''**, and registers it with the component. **Y** is the default. When **SENDEVENT='N'**, no Event name will be registered

SetCAM=*method-name*
specifies the custom access method to be executed when the attribute value is assigned.Using dot notation to set an attribute for which a setCAM method has been defined may result in side effects. See “What Happens When Attribute Values Are Set or Queried” on page 122.

State=**'N'** | **'O'**
determines whether the attribute is new or is overridden. **N** is the default.

ValidValues=*valid-values*
specifies the values that are valid for the CHARACTER attribute. Use a space or '/' or ',' to separate the values.
The following options are used for compatibility with Version 6 classes:

Automatic=**'Y'** | **'N'**
specifies whether *var-name* is an automatic instance variable.

IV=*V6-instance-variable-name*
specifies the name of a Version 6 instance variable.

PureIV=**'Y'** | **'N'**
When **PureIV='Y'**, it specifies that *var-name* is a pure Version 6 instance variable and that no associated Version 8 attribute will be created. **N** is the default.

method-declaration-statements
list the class methods.
For *method-declaration-statements*, use the following syntax:

```
method-label-name : <access-scope> METHOD<parameter-list>
                  < / (method-options)>;
```

method-label-name
can be up to 32 characters and has the same restrictions as an SCL label. By default, you should treat *method-label-name* the same as the method name. To define a more descriptive method name which is over 32 characters, use the **method=** option.

access-scope
can be one of the following:

PUBLIC
designates a method that can be inherited by subclasses and accessed anywhere the corresponding object exists. This is the default.

PRIVATE
designates a method that can be accessed only by methods in the class in which the method is defined. Private methods will not be inherited by subclasses of the class.

PROTECTED
designates a method that can be accessed only by subclasses in which the method is defined. Since a class can be considered a subclass of itself, a protected method can also be accessed from the class in which it is defined.

parameter-list
For parameter options such as using Input/Output/Update to store the parameter storage, using “**:**” operator to specify the parameter type, using **Optional=** to specify the varying arguments in the method, and using **Return=** to specify the

method return type, as well as **Arglist=** and **Rest=**, can all be applied in the parameter list. For more information, see "METHOD" on page 540.

method-options
specify options for a class method. You must put the list inside parentheses that follow a / (slash). Separate multiple options with commas. The available options are

Abstract=**'N'** | **'Y'**
specifies that the method is an abstract method and does not have an implementation associated with it. Abstract methods can be declared only in abstract classes. The default is **'N'**.

Description=*method-description-string*
specifies the description of the method.

Enabled=**'N'** | **'Y'**
determines whether a method can be temporarily disabled. **Y** is the default.

Label=*'method-label'*
identifies a method whose label is different from the method-label-name. If the **Label=** option exists, the **Method=** option cannot be used.

Method=*'method-name'*
identifies the *method-label-name* as the label name and the 'method-name' will be used for the method reference in the dot notation or CALL SEND routine. Since the '*method-name*' is a string, you can extend the method name up to 256 characters. If the **Method=** option exists, the **Label=** option cannot be used.

Native=*'/executable-name:n*
specifies the name of a system-implemented method.

Note: This option is generated by the CREATESCL function. △

SCL | Entry=*four-level-entry-name-string*
identifies the entry that contains the USECLASS block that implements the method. This option is required when the method is not implemented in the SCL entry that contains the CLASS statement block.

Signature=**'N'** | **'Y'**
determines whether the method has a signature. Y is the default. All methods in Version 6 classes have **Signature='N'**. Adding *parameter-list* and removing **Signature='N'** for each method will cause the SCL compiler to generate signatures for that method. **Signature='Y'** is required for method overloading.

State=**'O'** | **'N'**
determines whether the method has been overridden or is new.

Forward=**'N'** | **'Y'**
determines whether the method can be forward referenced when **Forward='Y'**. The SCL compiler is a one-pass compiler and will report errors when referencing a method that has not been defined in the current class. Using **Forward='Y'** will allow the SCL compiler to suppress the error messages and delay validation of the forward methods which are required to be defined later in the current class. If the forward method is not defined before the ENDCLASS statement, the SCL compiler will report the error. **N** is the default. This option can be used for methods calling each other.

ArgDesc1 | ArgDesc2 |...| ArgDescN =*each-argument-description-string*
specifies each argument description. This option is used for documenting the parameters.

ReturnDesc=*return-argument-description-string*
specifies the return argument description. This option is used for documenting the return parameter.

method-implementation-blocks
contain any SCL statements for the defined methods. These statements perform the operations of the method.

event-declaration-statements
define the class events. Declare the events as follows:
EVENT *event-string-name* < / (*event-options*)>;

event-string-name
is the name of the event you are declaring.

event-options
specifies options for the event. You must put the list inside parentheses that follow a / (slash). Separate multiple options with commas. Event options can be

Description=*event-description*
specifies the description of the event.

Enabled=**'N'** | **'Y'**
determines whether an event can be temporarily disabled. **Y** is the default.

Method=*string*
identifies the method that handles the event.

Send=**'Manual'** | **'After'** | **'Before'**
determines when the object sends the event.
After is the default.

eventhandler-declaration-statements
define the event handler to be executed after the events are triggered. The event handler is an SCL method that handles the event. Declare the event handler as follows:

EVENTHANDLER *eventhandler-name*< / (*eventhandler-options*)>;

eventhandler-name
is the name of the event handler of an SCL class method that you are declaring.

eventhandler-options
specifies options for the event handler. You must put the list inside parentheses that follow a / (slash). Separate multiple options with commas. Event handler options can be

Description=*eventhandler-description*
specifies the description of the event handler.

Enabled=**'N'** | **'Y'**
determines whether an event handler can be temporarily disabled. **Y** is the default.

Event=*event-name*
specifies the name of the event.

Method=*string*
identifies the method that handles the event.

Sender='_SELF_' | '_ALL_'
identifies the location of the sender to trigger the event handler. When **Sender='_SELF_'**, the event handler will only listen to events from the class itself. When **Sender='_ALL_'**, the event handler will listen to events from any other class. Using the method _addEventHandler, you can dynamically add a sender to trigger the event.

Details The CLASS statement enables you to use SCL to create a class and to define attributes, methods, events and event handlers for the class. The CLASS block is especially useful when you need to make many changes to an existing class such as adding signatures to an existing class, or when you want to create a class in batch mode. Using the CLASS block provides the advantages of error detection at compile time and improved performance during run time. It also enables you to use short-cut notation. Instead of using _SELF_.*attribute* or _self.*method*(...) to reference the name of a class attribute or a class method, you simply specify the attribute or method name. This makes programs easier to read and maintain. In addition, you can overload method definitions, which means that multiple methods can have the same name, but have different numbers and types of parameters.

The program block that defines a class starts with a CLASS statement and ends with an ENDCLASS statement. A CLASS block can contain statements that define attributes, methods, events, event handlers and even METHOD statement blocks implementing the operations for methods. You can also put the METHOD statements that implement class methods in another SCL entry when you use the SCL= method option to specify the name of that entry. Then, in the SCL entry that is specified with SCL=, define the methods for the class within a USECLASS statement block. Defining methods in a separate entry is useful for enabling class methods to be created, tested, or maintained by multiple application developers. For more information, see "METHOD" on page 539.

To create a class from an SCL entry that contains a CLASS block, you must compile and save the SCL entry as a CLASS entry. To do this, either issue the SAVECLASS command or select

File ▶ Save Class

from the SCL Source Editor. This is equivalent to using the Class Editor to interactively create a CLASS entry. However, the Class Editor provides a graphical view of the class, whereas the CLASS statement in SCL provides a language view of the class.

Do not declare the _SELF_, _FRAME_, _CFRAME_, _METHOD_, or _EVENT_ system variables inside a CLASS or USECLASS block. SCL automatically sets these values when it is running methods that are defined in CLASS or USECLASS blocks. Redefining any of these system variables can introduce unexpected behavior.

In methods that are defined in a CLASS statement block, all references to the methods and the attributes of the class can bypass two-level references to _SELF_.*attribute* and _SELF_.*method*(...). Because these methods are defined within the class, the SCL compiler can detect whether an undefined variable is a local variable or a class attribute.

You can also use the _super method in method code inside a CLASS statement block without having to specify either an object identifier or the method whose super method you are calling. You can use the _super method to call any method. For example, to invoke the super ADD method, you would use

```
_super.add();
```

To override the _init method, you must first call the super _init method before overriding the _init method. For example:

```
_super._init();
   ...statements that define the
   overridden _init method ...
```

Any SCL function or routine can be called inside a METHOD statement block that is inside a CLASS block. Outside the METHOD statement block, only class attribute statements, event statements and event handlers are allowed in a CLASS block. Other than the IMPORT statement, no SCL statements can be written outside the CLASS block.

METHOD blocks can include labeled sections. However, labeled sections that are outside a method block must be re-coded as PRIVATE methods, and the LINK statements that call them must be changed to method calls. This programming style will make your applications more consistent with the object-oriented paradigm.

If a local variable that is defined in a METHOD block has the same name as a class attribute, SCL gives precedence to the local variable. If a class method has the same name as any SCL-supported function, SCL gives precedence to the function. If an attribute array has the same name as a class method, SCL gives precedence to the method. It is probably best to avoid using the same name for multiple local variables, class attributes, method names or arrays to avoid problems.

The CLASS statement also enables you to define method definitions for overloading methods, which means that multiple methods have the same name. Methods that have the same names are allowed in a CLASS block only if the signatures, or parameter numbers or types, are different. For example, a class can have one COMBINE method that has numeric parameters and adds parameter values, and another COMBINE method that has character parameters and concatenates parameter values.

Inheritance from multiple classes is not supported in class syntax, but is allowed with interface syntax. For more information, see “INTERFACE” on page 486.

Examples

Example 1: A CLASS Block with Method Implementation This example defines the Arith class, a subclass of Sashelp.Fsp.Object.class, and implements the methods in the CLASS entry. The example shows the METHOD statements using the RETURN= option and then RETURN statements returning values to the caller.

```
class work.classes.arith.class;
    public  num  total;
    public  char catstr;
        /* A method that adds numeric values */
    add:  public method n1:num n2:num return=num;
        total = n1 + n2;
        return (total);
    endmethod;
        /* A method that concatenates */
        /* character values           */
    concat: public method c1:char  c2:char return=char;
         catstr = c1 || c2;
         return (catstr);
    endmethod;
endclass;
```

Example 2: A CLASS Block without Method Implementation This example defines the Combine class and specifies the SCL entry in which the methods are implemented. The class uses overloaded COMBINE methods, one to process numeric values and another to process character values. The code that implements the methods is defined in a USECLASS block.

```
class work.classes.combine.class;
    public  num  total;
    public  char catstr;

    combine: public method n1:num  n2:num  return=num
       / (scl='work.classes.combine.scl');

    combine:  public method c1:char c2:char
       return=char
       / (scl='work.classes.combine.scl');
endclass;
```

Here is the USECLASS block that contains the method implementations for WORK.CLASSES.COMBINE.CLASS:

```
useclass work.classes.combine.class;
   combine:  public method
         n1:num  n2:num  return=num;
      total = n1 + n2;
      return (total);
   endmethod;

   combine: public method
          c1:char  c2:char return=char;
      catstr = c1 || c2;
      return (catstr);
   endmethod;
enduseclass;
```

Example 3: Definition of a Class This example imports the Collection class, which is provided with SAS/AF software, and shows several forms of attribute declarations, method declarations, and overloading methods. Attributes list1 and list2, which define SCL list initialization, can also be found in this example.

```
import sashelp.fsp.collection.class;
class work.classes.myclass.class
    extends sashelp.fsp.object.class
    / (description = 'my first class file');
       /* simple attribute with no options */
    public num num1;
       /* Attribute with options */
    public num num2
       / (description = 'second numeric attribute',
          initialvalue= 3,
          validvalues = '1 2 3');
       /* Another attribute with options */
    public char string1
       / (editable = 'n', linkable = 'n',
         initialvalue = 'abc');

 /* SCL List initializations:items without name*/
    public list list1
 / (InitialValue={1, 2, 3, 'abc', 'def', 4, 5, 6}
   );
```

```
  /* SCL List initializations:Items with name.*/
  /* Address is a sublist of list2          */
     public list list2
  / (InitialValue={name='John Doe', Number=100,
     Address={State='NC', CITY='CARY'},
     Office='Bldg Z'} )
     /* Private array attribute  */
     private num arr(3) ;
        /* Private list attribute   */
     private list list;
        /* Protected collection attribute */
     protected collection coll;

        /* public method m0 */
    m0:  Public method
             /* External method implementations */
        / (scl='mylib.classes.b.scl',
          label = 'M0',
          description='test method m0');

        /* Public method m1        */
        /* with no method options  */
    m1:  public method ;
       ...more SAS statements...
    endmethod;

       /* Private overloading method m1 */
       /* with numeric parameter        */
    m1:  private method n: num;
       ...more SAS statements...
    endmethod;

       /* Protected overloaded method m1.
        * Method implementations should be placed in
        * work.classes.c.scl */
    m1:  protected method s: char
      /* external method implementation */
       / (scl = 'work.classes.c.scl');

       /* Other public method */
    m2: method return=num;
       ...more SAS statements...
       return (1);
    endmethod;

        /* Private method */
    m3: private method;
       ...more SAS statements...
    endmethod;
 endclass;
```

Example 4: Illustrating Short-Cut References This example shows how to use the _super method as well as short-cut references to _SELF_:

```
CLASS work.classes.another.class;
    Public Num n1;
    Public Num n2;
    Public Char c1;
    Public Char c2;
    Public Num m(3);

    _Init:  method / (State='O');
        DCL Num  n2;
        DCL Char c2;
            /* Equivalent to call super(_init); */
        _SUPER();
           /* Equivalent to _SELF_.N1 = 1   */
        n1 = 1;
           /* Local variable N2 dominates class */
           /* attribute N2                      */
        n2 = 1;
        m{1} = abs(n2);
           /* Uses { to avoid ambiguity    */
           /* Equivalent to _SELF_.C1 = 'a' */
        c1 = 'a';
           /* Local variable C2 dominates */
           /* class attribute C2          */
        c2 = 'a';
     endmethod;
        /* commonly used method can be PRIVATE */
     Common: Private Method a:Num;
        ...more SCL statements...
        a = a + 1;
     endmethod;

     M: method;
          /* Equivalent to              */
          /*    if _SELF_.N1 > 0 then   */
       if n1 > 0 then
            /* Equivalent to      */
            /*     _SELF_.N1 + 1; */
          n1 + 1;
       common(n1);
     endmethod;
           /* Method M1 with numeric parameter */
     M: method n: Num;
           /* Equivalent to _SELF_.M(); */
        M();
        common(n1);
     endmethod;

endclass;
```

Example 5: Illustrating Set Custom Access(setCAM) Method This example shows a setCAM method, M1, which will be invoked when the attribute N is modified in the _init method.

```
Class work.mycat.camDemo.class;
    Public num n / (initialValue = 5,
                  setCam='m1');
    _init: Method / (State='O');
           _super();
           n = 3;
           EndMethod;
    m1:    Method nParm:Num;
     /* - nParm is input value of attribute n */
           nParm = nParm + 100;
     /*  nParm is output value of attribute n */
           EndMethod;
EndClass;
```

To reference the camDemo class, you can code the following program:

```
Init:
    DCL  camDemo  obj = _new_ camDemo();
    obj.n = 7;
    /*   Using the SCL debugger to trace the sequence
     *   of this program, you will find the value
     *   of obj.n = 107 */
    put obj.n=;
    Return;
```

Example 6: Illustrating User-defined Events and Event Handlers This example shows a system-defined event, 'n Changed', which will be automatically generated to associate with the attribute N. An event handler, M1, which is associated with the 'n Changed' event is also written and will be executed when the attribute N is modified. Another user-defined event, 'myEvent', is created by the EVENT statement. The associated event handler, M2, will be executed when an explicit _sendEvent method is executed.

```
Class work.mycat.eDemo.class;
     Public num n; /* sendEvent='Y' is default*/
     Public char c / (sendEvent='N');
     Event 'myEvent'
        / (method='m2');
     EventHandler m1
        / (Sender='_SELF_',
           Event='n Changed');
     EventHandler m2
        / (Sender='_ALL_',
           Event='myEvent');
     m1: method a:list;
       Put 'Event is triggered by attribute N';
         endMethod;
     m2: method a:string b:num ;
       Put 'Event is triggered by _sendEvent';
         return (100);
         endMethod;
EndClass;
```

You could use the following program to trace the control sequence of the event handler by using the SCL debugger.

```
 init:
    DCL eDemo obj = _new_ eDemo();
    obj.n = 3;  /* will trigger the m1 */
```

```
      obj._sendEvent('myEvent','abc',3);
      Return;
```

Example 7: Illustrating Forward Method, Optional= and ArgList= This example shows how to use the method option Forward='Y' to write a method, M1, which can invoke another method, M2, defined after the current method, M1. Without the Forward='Y' option, the SCL compiler will issue an error. The M1 method contains Optional=, which actually includes two overloading methods.

```
Class mylib.mycat.forward.class;
   m2: Method n:num c:char Return=Num / (Forward='Y');
   m1: Method n1:num Optional=n2:num Arglist=argList
Return=Num;
          DCL Num listLen = listlen(argList);
          DCL Num retVal;
          if (listLen = 1) then
             retVal = m2(n1, 'abc');
          else if (listLen = 2) then
             retVal = m2(n2, 'abc');
          Return(retVal);
       EndMEthod;
   m2: Method n:num c:char  Return=Num;
          Return(n+length(c));
       EndMethod;
EndClass;
```

See Also

"ARRAY" on page 254

"CREATESCL" on page 316

"ENDCLASS" on page 366

"METHOD" on page 540

"USECLASS" on page 698

CLEARLIST

Clears the items from an SCL list without deleting the list and optionally clears all sublist items

Category: List

Syntax

rc=**CLEARLIST**(*list-id*<,*recursively*>);

rc

indicates whether the operation is successful:

0	successful

≠0	not successful

Type: Numeric

list-id

is the identifier of the list that contains the items to clear. An invalid *list-id* produces an error condition.

Type: Numeric or List

recursively

indicates whether to recursively clear all the list's sublists as well as all sublists of its sublists.

'N'	Sublists are not cleared. (This is the default.)
'Y'	All sublists are cleared.

Type: Character

Details CLEARLIST clears all the items from the SCL list identified by *list-id*. The list is not deleted, but its length is reduced to 0. If *recursively* is **'Y'**, then all the list's sublists, including sublists of sublists, are also cleared.

CAUTION:

Before you clear a list recursively, make sure it is not needed by other parts of the SCL program. The *recursively* option clears all of a list's sublists, even if they are referenced in other SCL lists or by other SCL variables. △

An error condition results if

- the list has the NOUPDATE or FIXEDLENGTH attribute.
- any item in the list (or in its sublists, if *recursively* is **'Y'**) has the NODELETE attribute.
- *recursively* is **'Y'**, and any sublist has the NOUPDATE or FIXEDLENGTH attribute.
- *list-id* is a component or class identifier.

If an error condition results, no further items or sublists are cleared.

Example

Clear all sublists from an existing list identified by MYLIST without deleting items that are not sublists:

```
   /* Copy the list.  */
cp=copylist(mylist);
  /* Clear the entire list, including sublists  */
  /* that also appear in CP.                    */
rc=clearlist(mylist,'Y');

   /* Copy the old list data.  */
   /* Sublists have been cleared.  */
mylist=copylist(cp,'N',mylist);

   /* Delete the copied list. */
rc=dellist(cp);
```

See Also

"COPYLIST" on page 311

"DELITEM" on page 335

"DELLIST" on page 336

"DELNITEM" on page 337

"SETLATTR" on page 656

CLOSE

Closes a SAS table

Category: SAS Table

Syntax

sysrc=**CLOSE**(*table-id*);

sysrc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

table-id
is the identifier that was assigned when the table was opened. A *table-id* value of **-999** closes all tables that were opened with OPEN. If *table-id* is invalid, the program halts.

Type: Numeric

Details Close all SAS tables as soon as they are no longer needed by an application. You do not need to open and close a SAS table in each program of an application. If an application contains several programs that use the same table, the first program can open the table and can use the parameter passing mechanism with the DISPLAY routine or method calls to make the table identifier value available to other programs.

Example

Use OPEN to open a SAS table. If the table opens successfully, as indicated by a positive value for the PAYID variable, then use CLOSE to close the PAYROLL table.

```
payid=open('payroll','u');
   ...SCL statements...
if (payid>0) then payid=close(payid);
```

See Also

"OPEN" on page 576

CLRFLD

Clears the value from variables whose values match a specified value

Category: Control or Field

Syntax

CALL CLRFLD(*pattern*,*variable-1*<, . . . ,*variable-10*>);

pattern
is the character string to match.
Type: Character

variable-1, . . . , variable-10
names up to ten character variables. If the value of a variable in this list matches *pattern*, then that value is cleared.
Type: Character

Details Variables in the variable list whose values do not match *pattern* exactly are not changed. No error occurs if there are no matches.

Example

Clear the value of any variable in the group SYM1 through SYM5 whose value is **BLUE**:

```
call clrfld('blue',sym1,sym2,sym3,sym4,sym5);
```

See Also

"SETFLD" on page 650

COLORLIST

Displays a selection list of the names of a device's valid colors and returns user selections

Category: Selection List

Syntax

selections=**COLORLIST**(*color-set*,*num-sel*< ,*message* < ,*autoclose*>>);

selections
contains one or more user selections from the list. Multiple selections are separated by blanks. By default, *selections* is 200 bytes long. To accomodate values longer than 200 bytes, you should explicitly declare *selections* with a longer length.
Type: Character

color-set
specifies the set of colors to display in the selection list:

device	specifies the name of a SAS/GRAPH device. If *device* is supplied, the selection list includes only the colors that are valid for the specified device, and an **All...** choice to display all possible colors. *Device* can be the name of a monitor, plotter, printer, or camera. This name can be up to eight characters long and must be specified within quotes. If the device entry is not found, the list contains all possible colors without regard to whether the device supports them.
'?'	opens the Color Selector window in which a user can design a color. Only one color can be defined, so *num-sel* is ignored. For additional information, use the window's online help.

Type: Character

num-sel
is the maximum number of items that a user can select from the list. To display the list for information purposes only (no selections allowed), specify **0**. To specify an unlimited number of selections, use a value such as 9999 that is larger than the number of available selections. A user cannot make a number of selections that exceeds the number of items in the list.
Type: Numeric

message
is the message text to display above the selection list. The default message tells users to make up to the number of selections specified in *num-sel*.
Type: Character

autoclose
specifies whether the selection list window closes automatically after a user makes a selection when only one choice is allowed:

'Y'	closes the window automatically. (This is the default.)
'N'	leaves the window open until the user explicitly closes it.

This option is ignored when *num-sel* is not 1.
Type: Character

Details By default, the message above the selection list asks the user to make *num-sel* selections. Also by default, the selection list window closes when the user makes a choice and presses ENTER if the selection list allows only one choice.

The FIND item in the selection list window enables you to specify characters that a value in the list contains. If the value is found, it is displayed as selected. If the value is not found, a message is displayed in the selection list window.

You can provide default values that will be initially selected when the color selection list is displayed. To do this, assign the values to the *selections* variable before calling COLORLIST.

If a user closes the selection list window without making a selection, COLORLIST returns a blank value unless there was an initial value for the *selections* variable before COLORLIST was called.

Selections from the window can be placed in the current result list, if one is available. The current result list is a special SCL list that is automatically filled with the values that are selected from a selection list. To use a current result list, use the MAKELIST function to create it, and use the CURLIST function to designate it as the current result list. The current result list must exist before you call the COLORLIST function.

You can use COLORLIST to enable a user to interactively design the RGB components for a color. If a user designs a color that is not supported on that device, SCL uses the closest color that is supported.

When COLORLIST is invoked, the current result list is cleared. After COLORLIST is invoked, the result list contains one element for each selected color name. The selections can be retrieved by using GETITEMC.

Example

Display a list of devices of type MONITOR that are available in the catalog SASHELP.DEVICES and allow users to select a device. Users can choose up to four colors from the selection list. If no device is chosen, display a list of all possible colors.

```
usrdev=devlist('sashelp.devices','monitor',
                       'Select a device. ');
device=substr(usrdev,41,8);
devcolor=colorlist(device,4);
```

Use a current result list to process multiple selections:

```
listid=makelist();
rc=curlist(listid);
selection=devlist('sashelp.devices','monitor',
                           'Select a device.');
device=substr(selection,41,8);
devcolor=colorlist(device,4);
n=listlen(listid);
do i=1 to n;
   color=getitemc(listid,i);
   put color=;
end;
```

Display a color selection dialog window:

```
color=colorlist('?',1,'Design a color for the
                                         bars');
```

See Also

"CURLIST" on page 320

"DEVLIST" on page 342

COMAMID

Returns the list of communications access methods for an operating system

Category: SAS System Option

Syntax

comamids=**COMAMID**(*options*);

comamids
are the communications access methods (comamids) for the operating system or for SAS/SHARE software or SAS/CONNECT software, if they are requested. Multiple values are separated by blanks.
Type: Character

options
requests comamid values that are supported by SAS/SHARE software or SAS/CONNECT software for your operating system:

'S'	requests comamid values that are supported by SAS/SHARE software.
'C'	requests comamid values that are supported by SAS/CONNECT software.

Details COMAMID provides a list of communication access method values for a user's operating system. If no value is provided for *options*, then all comamid values for the operating system are returned.

You can display the list to application users by using other SCL features. For example, you can display the values in a list box by specifying that the source of list box values is the variable that you used as the return value for COMAMID.

Note: COMAMID verifies communication access method values so that if the module to support a value has not been installed, that value is not returned in the string of comamid values. △

Examples

Example 1: Finding the Comamids for an Operating System Find out which comamids are valid for the operating system:

```
INIT:
   comamids=COMAMID();
   put comamids=;
return;
```

This example produces the following output on an HP-UX system:

```
COMAMIDS= TCP
```

Example 2: Finding Comamids for SAS/SHARE Software Find out which comamids are supported by SAS/SHARE software:

```
INIT:
   comamids=COMAMID('S');
   put 'Comamids for SAS/SHARE are 'comamids=;
return;
```

This example produces the following output on an HP-UX system:

```
Comamids for SAS/SHARE are COMAMIDS=TCP
```

COMPAREARRAY

Allows you to compare two arrays for size and data equality

Category: Array

Syntax

rc=**COMPAREARRAY**(*array1*,*array2*);

rc
indicates whether the two arrays match.

0	arrays match
1	arrays do not match

Type: Numeric

array1
is one of the two arrays to be compared.
Type: Array

array2
is one of the two arrays to be compared.
Type: Array

Details The COMPAREARRAY function allows you to compare two arrays for size and data equality. To be considered equal, the arrays must:

- have the same number of dimensions
- be of the same type
- have the same bounds
- have the same values in each element.

Examples

Example 1: Compare Arrays The example compare several different arrays.

```
DCL num n1(5) n2(5,5) n3(*) n4(5) n5(*,*);
DCL char c1(5);
DCL num rc;

rc = comparearray(n1,n2); put rc=;
rc = comparearray(n1,c1); put rc=;
rc = comparearray(n1,n3); put rc=;
rc = makearray(n3,3);
rc = comparearray(n1,n3); put rc=;
rc = redim(n3,5);
rc = comparearray(n1,n3); put rc=;
do i=1 to 5;
   n1[i] = i;
   n4[i] = i;
end;
```

```
rc = comparearray(n1,n4); put rc=;
rc = copyarray(n2,n5);
rc = comparearray(n2,n5); put rc=;
rc = delarray (n3); rc delarray (n5);
```

The output for this code would be:

```
rc=1
rc=1
rc=1
rc=1
rc=0
rc=0
rc=0
```

See Also

"COPYARRAY" on page 309

"DELARRAY" on page 332

"MAKEARRAY" on page 533

Chapter 4, "SCL Arrays," on page 37

COMPARELIST

Compares two SCL lists

Category: List

Syntax

rc=**COMPARELIST**(*list1-id*,*list2-id*<,*options*>);

rc
contains the return code for the operation:

0	The lists match.
1	The lists do not match.

list1–id, list2–id
contain the identifiers for the two SCL lists to be compared.

options
specify one or more comparison options. Use a space to separate multiple options. The available options are:

NAME (Default)|NONAME
determine whether the comparison is performed on the names of list items that are in the same position in both lists. NONAME does not compare item names.

NOHONORCASE (Default)|MIXEDCASE
determine whether the comparison is performed on the uppercase or mixed case values of all item values and names. NOHONORCASE compares the uppercase

values of item names and values. MIXEDCASE compares mixed case names and values.

ITEM (Default)|NOITEM
determine whether the comparison is performed on the values of list items that are in the same position in both lists. NOITEM does not compare list values.

NODUMP (Default)|LONGDUMP|SHORTDUMP
determine the extent to which differences are reported. NODUMP produces no messages. LONGDUMP displays all differences in the LOG window. SHORTDUMP displays the first five differences.

Details

COMPARELIST enables you to compare the information in two SCL lists. This comparison can include item names, values, or both. Names and items can be compared in mixed case.

Example

Compare the item names and values in OLDLIST and NEWLIST. If the lists are not the same, write all the error messages to the SAS log. If the lists are the same, delete OLDLIST.

```
rc=comparelist(oldlist,newlist,'name item nohonorcase
    longdump');
if rc=0 then
    rc=dellist(oldlist);
else do;
   ...SCL statements to run
   when the lists do not match...
end;
```

CONTENTS

Displays the attributes of a SAS table

Category: Catalog and SAS Table

Syntax

rc=**CONTENTS**(*SAS-table*<,*mode*>);

rc
contains the return code for the operation:

0	The attributes of the specified table were displayed.
≠0	An error or warning condition occurred during the operation.

Type: Numeric

SAS-table
is the name of the SAS table. (SAS data set options are ignored in this argument.) The name of the data set would have to include the #nnn number of the generation set. For example to view the third generation data set of WORK.ONE:

```
rc=contents('work.one#003');
```

Type: Character

mode
specifies whether the information can be modified:

`'B'`	displays the Properties window for browsing only.
`'E'`	allows information in the Properties window to be modified. (This is the default.) If member-level locking is not available, then the Properties window is displayed in BROWSE mode instead. *Note:* Any value that begins with a character other than **B** or **b** also selects EDIT mode. △

Type: Character

Details The CONTENTS function opens the Properties window, which enables an application user to browse or modify column names, formats, informats, and labels of a SAS table. By default, the Properties window is opened in edit mode. However, if the specified table is currently open, then you must specify **B** for *mode*.

Initially, General Properties (that is, attributes, but not column names) are listed in the Properties window. To change the value of an attribute, do the following:

1 Click the mouse menu button on the attribute that you want to change and then click on $\boxed{\text{Modify}}$.
2 In the dialog window that appears, make the desired changes to the text.

To change a column name, do the following:

1 Click on the down arrow at the upper right corner of the window. The menu that appears contains information about the table, including column names. Select the column name that you want to change.
2 Click the mouse menu button on the column name that you want to change, and then select $\boxed{\text{Modify}}$.
3 In the dialog window that appears, make the desired change to the column name.

Example

Display the attributes for the table MYLIB.HOUSES:

```
if (contents('mylib.houses')) then
   do;
      _msg_=sysmsg();
      ...SCL statements to handle case where
      contents cannot be displayed...
   end;
```

CONTINUE

Stops processing the current DO loop and resumes with the next iteration of that DO loop

Category: Control Flow

Comparisons: SAS Statement with limitations in SCL

Syntax

CONTINUE;

Details The CONTINUE statement is provided in SCL to control the execution of DO loops. When you need to force the statements in a DO loop to stop executing, you can use the CONTINUE statement to stop executing successive statements in a DO loop, and to move back up and re-execute the DO loop, starting with the header statement.

Note: In DATA step code, the CONTINUE statement stops processing only the current DO loop. In SCL code, the CONTINUE statement stops processing the current DO loop *or DO group*, whichever is closest. For example, suppose your code contains a DO loop that contains DO groups:

```
do n=1 to 5;     /* DO loop */
  if n=2 then do; continue; end;   /* DO group */
  put n=;
end;
```

When this code is compiled and run as part of an SCL program, the output is:

```
n=1
n=2
n=3
n=4
n=5
```

When this code is submitted as part of a DATA step, the output is:

```
n=1
n=3
n=4
n=5
```

See “DO” on page 356 for more information on DO groups and DO loops. △

When you use DO WHILE and DO UNTIL statements, use caution to prevent the CONTINUE statement from forcing the program into an infinite loop. For example, the following statements produce an infinite loop because the value of the variable I never exceeds 2. When I has the value of 2, the IF statement always causes a branch around the next two SCL statements.

```
   /* This example causes an infinite loop */
INIT:
i=1;
do while (i<1000);
   if mod(i,2)=0 then
      continue;
   sum+i;
   i+1;
end;
return;
```

See the documentation for the CONTINUE statement in *SAS Language Reference: Dictionary* for more information.

Example

Count the number of females in the SAS table WORK.PERSONEL and display their average age. WORK.PERSONEL contains the column GENDER, which contains the values **F** for female and **M** for male, as well as the column AGE, which contains numeric values for age. The display window contains two numeric controls: AVGAGE and FEMALES. If the value of GENDER is not **F** (female), then the CONTINUE statement skips the other statements and returns to the DO WHILE statement to read the next row. The results are displayed in the application window, although the individual records are not displayed.

```
INIT:
   females=0;
   total=0;
   persnlid=open('personel');
   call set(persnlid);
  /* Process rows until all the */
  /* rows are read. */
   do while (fetch(persnlid) ne -1);
         /* Skip males when processing. */
      if gender ne 'F' then
         continue;
      females+1;
      total+age;
   end;
 /* Display the results in the fields */
 /* FEMALES and AVGAGE. */
      avgage=total/females;
return;
MAIN:
  ...other SCL statements...
return;

TERM:
   rc=close(persnlid);
return;
```

See Also

"DO" on page 356

"LEAVE" on page 504

CONTROL

Controls the execution of labeled program sections and the formatting of submit blocks

Category: Control Flow

Comparisons: SAS Statement with limitations in SCL

Syntax

CONTROL *options*;

options

specify the type of control for program statements. The available options are described below. You can use one or more options.

Type: Character

ALLCMDS | NOALLCMDS

NOALLCMDS is in effect by default. Global or procedure-specific commands execute immediately without executing the SCL program. The program cannot intercept any procedure-specific commands that are issued in the application. Use CONTROL ALLCMDS to enable SCL to intercept procedure-specific or custom commands that are issued in the application. You can use the NEXTCMD routine to ignore invalid commands. Use CONTROL NOALLCMDS to restore the default behavior.

ALLCMDS provides the same functionality as the ALWAYS option and enables a program to intercept custom commands. In addition, ALLCMDS allows an SCL program to intercept procedure-specific commands. In FRAME entries only, ALLCMDS allows an SCL program to intercept full-screen global commands. For a listing of full-screen global commands, see "EXECCMDI" on page 381 .

In PROGRAM entries, ALLCMDS combines the effects of ENTER and ERROR and forces statements in the MAIN section to execute even if a user issues commands that are not recognized by the procedure.

In FSEDIT applications, ALLCMDS and ALWAYS have the same functionality, and both enable an SCL program to intercept any procedure-specific or custom commands.

When ALLCMDS is specified, statements execute in the MAIN section before a command that is issued with the EXECCMD routine. This behavior could introduce an infinite loop. Either execute the EXECCMD routine conditionally or specify the command using EXECCMDI with the NOEXEC parameter.

FSVIEW applications ignore these options.

ALWAYS | NOALWAYS

NOALWAYS is in effect by default. The MAIN section executes only when a user modifies a window variable with a valid value and then presses either ENTER or a function key. Use CONTROL ALWAYS to force statements in the MAIN section to execute even if a user issues commands that are not recognized by the procedure. ALWAYS combines the effects of ENTER and ERROR.

ALWAYS can be used if your application supports custom commands. When ALWAYS is specified, FSEDIT applications execute statements in the MAIN section before handling a command that is issued with the EXECCMD routine. This behavior could introduce an infinite loop. Either execute the EXECCMD routine conditionally or specify the command using EXECCMDI with the NOEXEC parameter.

FSVIEW applications ignore this option.

ASIS | NOASIS

NOASIS is in effect by default. SCL formats submit-block code and eliminates unnecessary spaces and line breaks. Use CONTROL ASIS so that submit blocks are submitted without formatting. You must use this option when the position of elements in the SAS code is important (for example, if you are submitting a DATALINES statement with a DATA step). ASIS is more efficient than NOASIS because it reduces the time spent on formatting.

BREAK *label* | NOBREAK

NOBREAK is in effect by default. If a program interrupt occurs while the SCL statements are executing, a requestor window opens and asks the user whether program execution should resume (that is, ignore the interrupt) or the program should quit. If a user chooses to quit execution, no more statements are executed for the current program, and control returns to the calling program. Use CONTROL BREAK so that you can specify a labeled section to which control passes if an interrupt or break condition occurs while the SCL statements are executing. *Label* is the program label of the section to execute after the current statement finishes execution. This labeled section can include SCL statements that report status and handle the interrupt. Use the _STATUS_ system variable to control execution such as **H** to halt and **R** to resume.

A program can contain any number of CONTROL BREAK statements. For example, there can be one in each of the INIT, MAIN, and TERM sections or in any other labeled section. When a CONTROL BREAK statement executes, any previous CONTROL BREAK statement is overwritten so that only one is in effect at a time.

Use NOBREAK to restore the default behavior. NOBREAK clears the current CONTROL BREAK specification.

FSVIEW applications ignore this option.

HALTONDOTATTRIBUTE/NOHALTONDOTATTRIBUTE

HALTONDOTATTRIBUTE is in effect by default. If SCL detects an error in the dot notation used in your application, the application halts execution. If you specify NOHALTONDOTATTRIBUTE, the application will continue to execute when errors are detected in the dot notation, but the results will be unpredictable. See “Accessing Object Attributes and Methods With Dot Notation” on page 119 for more information.

ENDSAS/NOENDSAS

NOENDSAS is in effect by default. When a user issues the ENDSAS or BYE command, the TERM sections of SCL programs in the current execution stack do not execute. This is called an ENDSAS event. Use CONTROL ENDSAS to force execution of the TERM section in an SCL program when an ENDSAS event occurs. ENDSAS has no effect on the _term method.

ENDAWS/NOENDAWS

NOENDAWS is in effect by default. When a user ends a SAS session by selecting the system closure menu in a FRAME entry that is running with the Application Work Space (AWS), the TERM sections of SCL programs in the current execution stack do not execute. This is called an ENDAWS event. Use CONTROL ENDAWS to force execution of the TERM section in an SCL program when an ENDAWS event occurs. ENDAWS has no effect on the _term method.

ENTER | NOENTER

NOENTER is in effect by default. The MAIN section executes only when a user modifies the value of a window variable and then presses either ENTER or a function key that is recognized by the procedure. Use CONTROL ENTER to force MAIN to execute when a user presses the ENTER key or a function key without modifying a window variable.

In FSVIEW applications, this option has an effect only if the cursor is on a valid row when ENTER or a function key is pressed.

ERROR | NOERROR

NOERROR is in effect by default. Statements in MAIN do not execute if a control or field contains a value that causes an attribute error. Thus, some statements in MAIN do not execute if multiple fields are in error and a user has not modified all of these fields. Use CONTROL ERROR to force statements in MAIN to execute even if the window contains fields that are in error.

If you use ERROROFF to remove the error status from a continued portion of a field in an FSEDIT application, then you must also use a CONTROL ERROR statement in the program. If a user does not type in the continued portion of the field and the program does not have a CONTROL ERROR statement, the error flag is not removed from the continued portion of the field. As a result, the default error message may be displayed, saying that a data value is not valid.

LABEL | NOLABEL
NOLABEL is in effect by default for PROGRAM and SCREEN entries. MAIN executes after any window variable is modified. Use CONTROL LABEL to force sections that are labeled with a window variable name (called window variable sections) to execute before MAIN executes. For FRAME entries, CONTROL LABEL is the default.

Statements in a window variable block execute after the associated window variable is modified, but only if the value does not introduce an error. That is, the value must satisfy any attributes that have been defined for the window variable.

Statements in MAIN do not execute until statements in all the window variable sections for modified fields execute successfully. The sequence for executing window variable sections is determined by the physical position of the field in the window from left to right and from top to bottom.

If a field modification introduces an attribute error, the associated window variable section does not execute. However, other window variable sections for modified window variables do execute. To correct an attribute error, you can allow users to correct the error in the window, or you can include SCL statements that make a correction in the labeled section for other fields.

If ERROR, ALWAYS, or ALLCMDS is also specified, then MAIN executes after the window variable sections even if an error was introduced.

If the window contains an extended table, the window variable section for each modified window variable executes for a row before the putrow section executes. MAIN executes after the putrow section executes.

Note: If CONTROL LABEL is specified, a window variable section must not contain a SUBMIT IMMEDIATE block. △

TERM | NOTERM
This option is valid only for FSEDIT applications. NOTERM is in effect by default. Statements in the TERM section of FSEDIT applications do not execute when a user scrolls off the current row in a SAS table unless the user changed the values of one or more columns so that the current row needs to be updated in the table, or unless the row is new. Use the TERM option to force execution of the statements in the TERM section even if a user does not modify any columns in the current row.

Details The CONTROL statement controls the execution of labeled program sections and also controls the formatting of code in a submit block. A CONTROL statement option remains in effect until another CONTROL statement option overrides it. Multiple CONTROL statement options can be in effect at the same time.

Examples

Example 1: Using the ASIS Option

Use the ASIS option:

```
control asis;
submit;
   data a;
      input x y z;
      datalines;
      10 20 30
      40 50 60
   run;
endsubmit;
rc=preview('display');
```

With the CONTROL ASIS statement in effect, the submit block executes without errors. If you remove the CONTROL ASIS statement, SCL formats the code within the block as follows when the code is submitted for processing:

```
   data a;
   input x y z;
   datalines;
   0 20 30 40 50 60 run;
```

When formatted in this manner, the final statement contains a syntax error and the code cannot execute properly.

Example 2: Controlling a Program Interrupt Define a break handler section labeled STOPINIT. When a user interrupts processing while SCL statements in INIT are executing, the STOPINIT label executes. If the loop index I is less than 350, execution of the program halts and control returns to the calling program. Otherwise, execution resumes. After the first loop is finished, execute CONTROL NOBREAK so that there is no break handler. If a user interrupts processing during the second loop, the SCL Break requestor window is displayed, and the statements in STOPINIT do not execute. The user can either abort or resume processing. Follow the same steps to define a new break handler section labeled STOPTERM in the TERM section.

```
INIT:
      /* define break label STOPINIT        */
   control break stopinit;
      /* loop 500 times to allow interrupt  */
      /* checking with control break        */
      /* if user interrupts, statements in  */
      /* label STOPINIT execute */
   do i=1 to 500;
      put i=;
   end;

      /* reset so there is no break handler */
   control nobreak;

      /* loop 500 times to allow interrupt  */
      /* checking without control break     */
   do i = 1 to 500;
      if (int(i/25) eq (i/25)) then put i=;
   end;
return;
MAIN: return;
TERM:
      /* Define the new break label STOPTERM. */
   control break stopterm;

      /* Loop 500 times to allow */
      /* interrupt checking with control */
      /* break.  If user interrupts,     */
      /* statements in label STOPTERM    */
      /* execute.                        */
   do j=1 to 500;
```

```
      put j=;
   end;

      /* Reset so there is no break handler.  */
   control nobreak;

      /* Loop 500 times to allow */
      /* interrupt checking without control  */
      /* break.                          */
   do j = 1 to 500;
      if (int(j/25) eq (j/25)) then put j=;
   end;
return;

STOPINIT:
      /* HALT if loop counter is less than 350, */
      /* otherwise RESUME.                      */
      /* Report the current status.             */
   put i=;
   if (i < 350) then
      _status_ = 'H';
   else
      _status_ = 'R';
return;

STOPTERM:
      /* HALT if loop counter is less than 350, */
      /* otherwise RESUME.                      */
      /* Report the current status.             */
   put j=;
   if (j < 350) then
      _status_ = 'H';
   else
      _status_ = 'R';
return;
```

See Also

"WORD" on page 725

COPY

Copies a SAS table, view, catalog, or catalog entry

Category: Utility

Syntax

sysrc=**COPY**(*old-name*,*new-name*<,*type*>);

sysrc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

old-name
is the name of the SAS file or catalog entry to copy. This can be a one-, two-, or four-level name and can include SAS data set options.
Type: Character

new-name
is the new name for the SAS file or catalog entry. This must be a three- or four-level name if *type* is **'CATALOG'**. If a catalog entry is being copied, the function sets the entry type of the new entry to that of the old entry. You can also specify SAS data set options.
Type: Character

type
is the type of SAS file or catalog entry to be copied:

'ACCESS'
The member is an access descriptor that was created using SAS/ACCESS software.

'CATALOG'
The member is a SAS catalog or a catalog entry.

'DATA'
The member is a SAS data file. (This is the default.)

'VIEW'
The member is a SAS data view.
Type: Character

Details To copy a catalog entry, specify the complete four-level name of the entry for *old-name*, a three-level name for *new-name*, and **'CATALOG'** for *type*.

You can use the WHERE= data set option to copy only those rows that meet a WHERE subset to the new table.

The COPY function does not copy existing integrity constraints to the new SAS table. Use the ICCREATE function to define new integrity constraints.

If the SAS table that is being copied is indexed, then all indexes are rebuilt for the new SAS table. *New-name* is ignored when you use COPY to copy GRSEG catalog entries that were created using SAS/GRAPH software. A copied GRSEG entry will have either the same name as the original entry or, if an entry with that name already exists in the target catalog, a unique name generated by SAS software.

CAUTION:
This function can overwrite existing files. If a table or catalog with the specified new name already exists, COPY overwrites the existing table or catalog without warning. △

Example

Copy the SAS table DATA1 to DATA2 and copy WORK.TEMP.A.SCL to SASUSER.PROFILE.B.SCL:

```
if (copy('data1','data2')) then
   do;
```

```
         _msg_=sysmsg();
         ...SCL statements to handle the
                    error condition...
      end;
   rc=copy('work.temp.a.scl',
           'sasuser.profile.b.scl', 'catalog');
   if (rc) then
      do;
         _msg_=sysmsg();
         ...SCL statements to handle the
                     error condition...
      end;
```

Copy from the SAS table SASUSER.CLASS to WORK.CLASS only those rows in which the variable GENDER=**M**:

```
   if (copy('sasuser.class(where=(GENDER=''M''))',
            'work.class')) then
      do;
         _msg_=sysmsg();
            ...SCl statements to handle the
            error condition...
      end;
```

See Also

"DELETE" on page 334

"ICCREATE" on page 460

"RENAME" on page 612

COPYARRAY

Copies data from one array into another array

Category: Array

Syntax

rc=**COPYARRAY**(*source_array*,*target_array*<,*ignoresize*>);

rc

indicates whether the operation was successful.

0	successful
≠0	not successful

Type: Numeric

source_array

is the array to copy the values from.

Type: Array

target_array
is the array to copy the values into.
Type: Array

ignoresize
indicates whether to check for array sizes to be the same.

'Y'	tells SCL to check whether the source and target arrays are the same size.
'N'	tells SCL to ignore array sizes. (This value is the default.)

Type: Character

Details The COPYARRAY function allows you to copy data from one array (*source_array*) into another array (*target_array*). The arrays must have the same dimensions and size and be of the same type. The source array being copied from can be a static or dynamic array, but if it is a dynamic array then its size must already have been set using MAKEARRAY or REDIM. The target array being copied into can also be a static or dynamic array. If it is dynamic and has not yet been created, then COPYARRAY will create the array to the same size of the source array. However, the low bound of a dynamic array is always 1, so the resultant target array may end up with different low or high bounds in each of its dimensions.

If you set *ignoresize* to `'Y'`, then the sizes of the arrays do not have to match. Only the types and dimensions of the arrays have to match. In this case, if the source array is bigger than the target array, then the elements in the source array will be truncated, and you will lose the data in the elements that do not coincide. If the source array is smaller than the target array, then the elements in target array that are not set will be automatically set to missing values.

If the COPYARRAY is used to create a dynamic array, the DELARRAY function should be used to delete the dynamic array.

Examples

Example 1: Copy Elements of a One–Dimensional Array The example copies the elements of the one–dimensional array A into the one–dimensional array B and prints out the contents of the arrays.

```
DCL num a(5) b(5);
do i=1 to 5;
  a[i] = i;
end;
rc = copyarray(a,b);
put a=; put b=;
```

The result of this code would be:

```
a=
a[1] = 1
a[2] = 2
a[3] = 3
a[4] = 4
a[5] = 5
b=
b[1] = 1
b[2] = 2
b[3] = 3
b[4] = 4
```

```
b[5] = 5
```

Example 2: Copy Elements of a Two–Dimensional Array The example copies the elements of the two–dimensional array A into the two–dimensional array B and prints out the contents of the arrays.

```
DCL num a(2,2) b(3,4);
count=0;
do i=1 to 2;
  do j=1 to 2;
    a[i,j] = count;
  end;
end;
rc = copyarray(a,b,'y');
put a=; put b=;
```

The result of this code would be:

```
a=
a[1,1] = 1
a[1,2] = 2
a[2,1] = 3
a[2,2] = 4
b=
b[1,1] = 1
b[1,2] = 2
b[1,3] = .
b[1,4] = .
b[2,1] = 3
b[2,2] = 4
b[2,3] = .
b[2,4] = .
b[3,1] = .
b[3,2] = .
b[3,3] = .
b[3,4] = .
```

See Also

"DELARRAY" on page 332

"MAKEARRAY" on page 533

"REDIM" on page 610

Chapter 4, "SCL Arrays," on page 37

COPYLIST

Copies or merges the contents of an SCL list into an existing list or a new list

Category: List

Syntax

new-list-id=**COPYLIST**(*list-id*<,*options*>
<,*target-list-id*>);

new-list-id
is either the identifier of the new list to contain a copy of the contents of *list-id*, if *target-list-id* is not supplied, or *target-list-id*, if a target list is supplied.
Type: Numeric

list-id
is the identifier of the list to copy or merge into the target list. An invalid *list-id* produces an error condition.
Type: Numeric or List

options
specify whether list values are merged and control how sublists are copied or merged. You can use one or more of the following values, separated by spaces. Later keywords override previous keywords.

'NONRECURSIVELY' | **'NO'** | **'N'**
copies or merges only sublist identifiers as values for sublist items. (This is the default.)

'MERGE' | **'M'**
merges the contents of the source *list-id* into the *target-list-id*, replacing like-named existing items in the target list. You may combine this option with the recursive option. An error occurs if *target-list-id* is not supplied or is not a valid list identifier.

'RECURSIVELY' | **'YES'** | **'Y'**
copies or merges all items of sublists and of sublists of sublists, and so on.
Type: Numeric

target-list-id
is the identifier of the list into which the source list is copied or merged. If supplied, *target-list-id* is also returned. Otherwise, a new list is created and returned. New sublists are created with the same environment (local or global) as the target list.
An error condition results if the target list has attributes such as NOUPDATE and FIXEDLENGTH that prevent copying data into it.
Type: Numeric

Details The copy operation appends items from the source list to the end of the target list, whereas the merge operation copies them into the target list, replacing existing named items.

If an SCL object is passed to COPYLIST as *list-id*, the resulting copy is not an SCL object. Although the new list contains all items from the original object, methods can not be called on this copy. The copied list is also treated as a regular list for comparisons (e.g., COMPARELIST).

If *target-list-id* is omitted, the function creates a new list in the same environment (L or G) as the list being copied and makes the new list the *target-list-id*. (For a description of list environments, see “ENVLIST” on page 373.) If *target-list-id* is supplied, its identifier is returned in *new-list-id*.

When a list is copied recursively, the items in all sublists are also copied, not just the sublist identifiers. However, even this duplication is avoided if it would result in an infinite recursion. When copying a list recursively, SCL does not perform an infinite recursive copy. For example, if a list contains itself, COPYLIST detects the circular structure and recreates the structure in the copy.

Merging occurs by item names. All items in the source list (and in its sublists, if merging is recursive) must have names. For each item, the name is used to find a matching name in the target list, as with NAMEDITEM(*list-id name*). If the same name appears multiple times in the source list, each item is merged independently. That is, the last occurrence of the name overwrites previous merged values and does *not* match with subsequent items in the target list. Thus, you should strive to keep item names unique in the source list in order to avoid wasted processing. If the corresponding item is not found in the target list, a new item is created.

In the merge operation, a list or sublist is merged only once, even if it appears multiple times. Also, a warning is printed for items that do not have names.

If an item in the source list has the NOWRITE attribute, then the corresponding item in the target list is *deleted*, unless it has the NODELETE attribute, in which case it is not merged. If a scalar item replaces a sublist item in a merge, the replaced list is not deleted because it may be used elsewhere. The SCL program must explicitly delete the old list.

All attributes of the list and its contents are preserved when a list is copied. The password is not copied so that you can modify the copy without knowing the password of the original list. The copy has no password. (See “SETLATTR” on page 656 for a discussion of passwords for lists.)

COPYLIST ignores any invalid options and uses its defaults instead.

Examples

Example 1: Copying a Single List

```
   /* make B a local named list */
   /* with 2 items named x, y   */
b=makenlist('L','x','y');
b=setnitemc(b,'ABC','x');
b=setnitemc(b,'XYZ','y');
   /* make A a local named list      */
   /* with 3 items named A, B, and C */
a=makenlist('L','A','B','C');
a=setnitemn(a,3.5,'A');
a=setniteml(a,b,'B');
a=setnitemn(a,9.75,'C');

call putlist(a,'A=',2);
NREC=copylist(a,'N');
   /* nonrecursive copy */
call putlist(NREC,'NREC=',2);
REC=copylist(a,'Y');
   /* recursive copy */
call putlist(REC,'REC=',2);
```

This program produces the following output:

```
A=(  A=3.5
      B=(  x='ABC'
           y='XYZ'
         )[3]
      C=9.75
    )[5]
NREC=(  A=3.5
        B=(  x='ABC'
             y='XYZ'
           )[3]
        C=9.75
      )[7]
```

```
REC=(  A=3.5
       B=(  x='ABC'
            y='XYZ'
          )[11]
       C=9.75
     )[9]
```

The sublist named B in the outer list NREC is the same list as the sublist named B in the outer list named A, from which NREC was copied non-recursively. Both lists named B have the same list identifier (3), which means they are in fact the same list. However, the sublist named B in the outer list REC, which was copied recursively from list A, is a different list, although it has the same contents as the list named B from A. The sublist in the outer list REC has a list identifier of 11, not 3, which shows it is a different list.

Note: [5], [7], and [9] are the list identifiers that were assigned when this example was run and may be different each time the example is run. △

Example 2: Appending a List to Itself Append the list MYLIST to itself. Both NEWLIST and MYLIST contain the list identifier for the copy of MYLIST.

```
mylist=makelist();
mylist=insertn(mylist,1,-1);
mylist=insertn(mylist,2,-1);
mylist=insertn(mylist,3,-1);
newlist = copylist(mylist,'N',mylist);
```

NEWLIST contains the values 1, 2, 3, 1, 2, 3.

Example 3: Merging One List into Another List

```
INIT:
   a = makenlist('L','A','B','C','D','E','F');
   do i = 1 to listlen(a);
      a = setitemc(a, nameitem(a,i),i);
   end;
   c = insertc(makelist(),'?',-1,'NOT');
   a = insertl(a, c,-1,'WHY');
   b = makenlist('L','A','E','I','O','U');
   do i = 1 to listlen(b);
      b = setitemn(b, rank(nameitem(b,i)),i);
   end;
   b =
   insertl(b, insertn(makelist(),0,-1,'NOT'),-1,
                                          'WHY');
   call putlist(a,'A before merge:');
   call putlist(b,'B before merge:');
   b = copylist(a,'yes merge',b);
   call putlist(b,'B after merge :');
return;
```

The result is

```
A before merge:(A='A' B='B' C='C' D='D' E='E'
                                           F='F'
WHY=(NOT='?' )[7] )[5]
B before merge:(A=65 E=69 I=73 O=79 U=85
WHY=(NOT=0 )[11] )[9]
B after  merge :(A='A' E='E' I=73 O=79 U=85
WHY=(NOT='?' )[11] B='B' C='C'
D='D' F='F' )[9]
```

The result list B contains items from A where the names intersect as well as original items from B for items that were not found in A. Because the sublist WHY was found in both, a recursive merge replaced 0 from the sublist of B with '?' from the sublist of A.

Note: 7, 5, 11, and 9 are the list identifiers that were assigned when this example was run and may be different each time the example is run. △

Example 4: Copying Multiple Instances of a List Copy a list, which contains a copy of itself, non-recursively and recursively. The outer list R1 contains two items, named SELF and R1, which are actually the same list as R1. When a non-recursive copy, R2, is made, the copy has items named SELF and R1 which are still the list R1. Only when R1 is copied recursively as R3 does the copy contain itself instead of R1.

```
/*  Create the list L, fill it, and print R1.  */
   r1=makenlist('l','a','SELF','r1', 'x');
   r1=setniteml(r1,r1,'SELF'));
   r1=setniteml(r1,r1,'r1'));
   r1=setnitemn(r1,1,'a'));
   r1=setnitemn(r1,99,'x'));
   call putlist(r1,'R1=',2));
/*  Copy R1 nonrecursively into R2 and print R2 */
   r2=copylist(r1,'n');
   call putlist(r2,'R2=',2);
/*  Copy R1 recursively into R3 and print R3 */
   r3=copylist(r1,'y');
   call putlist(r3,'R3=',2);
```

The list R2, which was created with a nonrecursive copy operation, contains the list R1. Note that the structure of the list R3 is identical to that of R1: it contains two copies of itself, at items named SELF and R1, because these items are lists whose list identifier is the same as the list R3.

This program produces the following output:

```
R1=(  a=1
      SELF=(...)[13]
      R1=(...)[13]
      x=99
    )[13]
R2=(  a=1
      SELF=(  a=1
              SELF=(...)[13]
              R1=(...)[13]
              x=99
            )[13]
      R1=(...)[13]
      x=99
    )[15]
R3=(  a=1
      SELF=(...)[17]
      R1=(...)[17]
      x=99
    )[17]
```

Note: 13, 15, and 17 are the list identifiers that were assigned when this example was run and may be different each time the example is run. △

Example 5: Merging Nonrecursively and Recursively Merge the contents of the list identified in SOURCEID into the list identified in TARGETID. The second call does a recursive merge.

```
targetid=copylist(sourceid,"MERGE",targetid);
targetid=copylist(sourceid,"MERGE YES",targetid);
```

See Also

"DELLIST" on page 336

"GETLATTR" on page 443

"HASATTR" on page 458

"MAKELIST" on page 535

"MAKENLIST" on page 536

"PUTLIST" on page 607

"SETLATTR" on page 656

CREATESCL

Writes class or interface information to an SCL entry

Category: Object Oriented

Syntax

entry-id=**CREATESCL**(*entry-name, scl-entry*<*,description*><*,proxy*>);

entry-id
contains the identifier that is assigned to the class, interface, or class package. If the class does not exist, *entry-id* contains 0.
Type: Numeric

entry-name
is the one- to four-level name of the CLASS, INTERFACE, or CLASSPKG catalog entry to load. If *entry-name* is a one- or two-level name, then the current search path is used to find the CLASS, INTERFACE, or CLASSPKG entry. If the entry was previously loaded, then the same identifier is returned. Otherwise, the entry is loaded from the catalog into the application class list, and the entry identifier is returned in *entry-id*.
Type: Character

scl-entry
is the one- to four-level name of the SCL entry to be created. If *proxy* is `1` and *entry-name* is a CLASSPKG entry, then you can specify a two-level name for *scl-entry*, and CREATESCL will generate a series of proxy entries for the classes in the class package.
Type: Character

description
is the description of the SCL entry.
Type: Character

proxy
specifies whether CREATSCL should generate proxy entries. The default value of **0** means that CREATESCL will not generate proxy entries.

Specify **1** if you want CREATESCL to generate proxies. If *entry-name* is a CLASS or an INTERFACE entry, then CREATESCL will generate a proxy entry for the class or interface. If *entry-name* is a CLASSPKG entry, and *scl-entry* is a two-level name, then CREATESCL will generate a proxy entry for each of the classes in the package (each class specified in an ITEM statement).
Type: Numeric

Details CREATESCL writes the class or interface definition that is in *entry-name* to an SCL entry. The class definition is written to a CLASS or INTERFACE statement block in the SCL entry that is specified in *scl-entry*.

You can revise the code in *scl-entry* and use the SAVECLASS command to create a new or revised class/interface. For example, you can use the CREATESCL function to generate a Version 8 class that contains Version 6 class information. You can add the method signatures, scope, and so on, and then generate a new class using the SAVECLASS command. This process is described in “Converting Version 6 Non-Visual Classes to Version 8 Classes” on page 131.

For more information on the SAVECLASS command, see the online Help for SAS/AF software.

For a description of the CLASS and INTERFACE statement syntax that is written to the SCL entry, see “CLASS” on page 277 and “INTERFACE” on page 486.

Examples

Example 1: Generate CLASS SCL Entry Generate the SCL code for SASHELP.CLASSES.CHECKBOX_C.CLASS. Store the SCL code in WORK.A.CHECKBOX_C.SCL:

```
myClass=CREATESCL('sashelp.classes.checkbox_c.class',
   'work.a.checkbox_c.scl',
   'CheckBox class to convert');
```

Example 2: Generate CLASSPKG SCL Entry Generate the SCL code for SASHELP.IOMSAMP.TSTPKG.CLASSPKG. Store the SCL code in WORK.A.B.SCL:

```
rc = createscl('sashelp.iomsamp.tstpkg.classpkg','work.a.b.scl');
```

Example 3: Generate A Proxy Entry Generate a proxy entry for SASHELP.CLASSES.SCLLIST.CLASS.

```
rc = createscl('sashelp.classes.scllist.class','work.a.c.scl','',1);
```

The CLASS statement needed to generate the proxy class is stored in WORK.A.C.SCL. If you issue the SAVECLASS command for WORK.A.C.SCL, then SCL will generate the associated proxy class SCLLISTPROXY.CLASS.

Note: CREATESCL will generate **native='/sasprxy:0'** for each of the methods defined in the class. Do not modify this method option. △

Example 4: Generate A Series of Proxy Entries Generate an SCL proxy file for each of the classes specified in SASHELP.IOMSAMP.TSTPKG.CLASSPKG. This class package contains a ITEM statement for these classes:

- □ SASHELP.IOMSAMP.AFTESTTYPES.CLASS
- □ SASHELP.CLASSES.SCLLIST.CLASS

```
rc = createscl('sashelp.iomsamp.tstpkg.classpkg','work.a','',1);
```

This statement creates two SCL entries in the WORK.A catalog: AFTESTTYPESPESPROXY.SCL and SCLLISTPROXY.SCL. These SCL entries contain the CLASS statements needed to generate the proxy classes. If you issue the SAVECLASS command for these SCL entries, SCL will generate the proxy CLASS files.

Note: CREATESCL will generate **native='/sasprxy:0'** for each of the methods defined in the class. Do not modify this method option. △

See Also

"APPLY" on page 252

"INSTANCE" on page 485

"LOADRES" on page 526

"NOTIFY" on page 572

"SEND" on page 644

"SUPAPPLY" on page 679

"SUPER" on page 681

CURCOL

Contains the value of the leftmost column in an extended table control in a FRAME entry

Category: System Variable

Details _CURCOL_ is a system variable. It is provided automatically by the FRAME entry in SAS/AF, and the SCL compiler automatically creates a space for it in the SCL data vector.

The value of _CURCOL_ is updated when the getrow or putrow section or the _getrow or _putrow method of an extended table is executing. Therefore, _CURCOL_ must be referenced only within these sections.

CURCOL is available only in SAS/AF FRAME entries.

Example

Suppose you have a text entry control, TEXT, in an extended table control. TEXT is assigned a value in the getrow section, based on a substring of a longer string. When the extended table is scrolled left and right, the value of _CURCOL_ is updated and is used as the *position* argument to the SUBSTR function.

```
GET1:
   text = substr( longstring, _curcol_ );
return;
```

For more information about extended table controls for horizontal scrolling, see the documentation for SAS/AF classes.

See Also

"_CURROW_" on page 322

CURFLD

Returns the name of the FRAME or PROGRAM entry control or field on which the cursor is currently positioned

Category: Cursor

Syntax

wvar-name=**CURFLD**();

wvar-name
contains the name of the FRAME or PROGRAM entry control or field on which the cursor is currently positioned.
Type: Character

Details The CURFLD function returns the name of the field FRAME or PROGRAM entry control on which the cursor is located. If the cursor is not positioned on a window variable, a null string is returned. CURFLD is usually used in conjunction with a CONTROL statement that includes the ENTER, ALWAYS, or ALLCMDS option. You can use CONTROL LABEL to achieve the same result more efficiently. FRAME or PROGRAM entries can also use the _getCurrentName method.

Example

Use CURFLD to control the behavior of an SCL entry application:

```
INIT:
   control enter;
return;

MAIN:
   select( curfld() );
      when('PHONE') call display('phone.help');
      when('EMPLOYEE') call display
                          ('employee.scl');
      otherwise;
   end;
return;
```

This example can be implemented without CURFLD if the program contains a program block that is labeled with the name of the window variable. If the program is in a SCL entry, then the window variables must be of type PUSHBTNC or PUSHBTNN, and the INIT section must contain CONTROL LABEL.

```
INIT:
   controllable;
   return;
```

```
PHONE:
   call display('phone.help');
return;
EMPLOYEE:
   call display('employee.scl');
return;
```

See Also

"CONTROL" on page 302

"CURSOR" on page 323

"CURWORD" on page 324

CURLIST

Designates or reports the current result SCL list

Category: List

Syntax

list-id=**CURLIST**(<*new-list-id*>);

list-id
is the identifier of the list to receive the values returned by the next SCL selection list function that is invoked.

>0	is the list identifier of the SCL list that was previously defined as current with the CURLIST function.
0	indicates that no list is defined as the current list.

Type: Numeric or List

new-list-id
is the identifier of the list to be designated as the current list. An invalid *new-list-id* produces an error condition.
Type: Numeric

Details When a value is provided for *new-list-id*, CURLIST designates the SCL list that is identified by *new-list-id* as the current result list. *New-list-id* must be the list identifier that was returned by the MAKELIST, MAKENLIST, or COPYLIST function that created this list. If you omit *new-list-id*, then CURLIST returns the identifier of the SCL list that was already designated as the current result list by CURLIST. If no argument is passed to CURLIST and no current result list has been specified, CURLIST returns 0.

The current result list is filled automatically with the values that are selected when the next SCL selection list function executes.

The functions that can fill the current result list are the SCL selection list functions CATLIST, COLORLIST, DATALISTC, DATALISTN, DEVLIST, DIRLIST, FILELIST, LIBLIST, LISTC, LISTN, LVARLEVEL, and VARLIST. For example, CATLIST opens a selection list window that displays the names of catalog entries. The value returned by CATLIST is a character string containing each catalog name that users select, separated by blanks. Selection list functions like CATLIST can also automatically fill the current result list, as specified by CURLIST, with a character item for each selection that users make. The length of that list is unbounded.

When one of the selection list functions is invoked, the values that users select replace the entire contents of the current result list. To preserve the contents of the current result list, use COPYLIST to copy the list before calling another selection list function.

Example

Set up a selection list, invoke a selection list function, and access the selections:

```
clist=makelist();
oldcurlist=curlist(clist);
   /* Allow user to choose up to 16 graphs.  */
graphs=catlist('SASUSER.DEMO','GRSEG',16,'Y');
n=getnitemn(clist,'COUNT');
do g=1 to n;
      graphName=getnitemc(clist,'name',g);
     put 'Selection #' g ' is ' graphName;
end;
/* Delete temporary curlist and restore +/
/* previous curlist. */
rc=dellist(clist);
oldcurlist=curlist(oldcurlist);
```

See Also

"CATLIST" on page 269

"COLORLIST" on page 293

"COPYLIST" on page 311

"DATALISTC and DATALISTN" on page 326

"DELLIST" on page 336

"DEVLIST" on page 342

"DIRLIST" on page 347

"FILELIST" on page 401

"LIBLIST" on page 513

"LISTC and LISTN" on page 518

"LVARLEVEL" on page 531

"MAKELIST" on page 535

"MAKENLIST" on page 536

"PUTLIST" on page 607

"VARLIST" on page 709

CUROBS

Returns the number of the current row in a SAS table

Category: SAS Table

Syntax

row-number=**CUROBS**(*table-id*);

row-number
is the current row number.
Type: Numeric

table-id
is the table identifier that was assigned when the table was opened. If *table-id* is invalid, the program halts.
Type: Numeric

Details

CAUTION:
This function should be used only with an uncompressed SAS table that is accessed using a native library engine. △

If the engine that is being used does not support row numbers, the function returns a missing value.

In FSEDIT and FSVIEW applications, specifying a value for *table-id* is optional. If the argument is not specified, CUROBS returns the value for the SAS table displayed by FSEDIT or FSVIEW. For a SAS table view, the function returns the relative row number. Also, on engines that do not support absolute row numbers, such as ORACLE, CUROBS returns the relative row number.

Examples

Example 1: Getting the Row Number of a Control Use FETCHOBS to fetch the tenth row in the SAS table MYDATA. CUROBS returns a value of 10 for *row-number*.

```
tableid=open('mydata','i');
rc=fetchobs(tableid,10);
rownum=curobs(tableid);
```

Example 2: Getting the Current Row Number In an FSEDIT SCL program, retrieve the number of the row that is currently displayed:

```
rownum=curobs();
```

CURROW

Contains the number of the current row in an extended table

Category: System Variable

Details _CURROW_ is a system variable that is created for every SCL program that you compile. The compiler creates a space for _CURROW_ in the SCL data vector.

CURROW is updated when the getrow or putrow section of an extended table is executing or when the _getrow or _putrow method of an extended table control is executing. Therefore, _CURROW_ must be referenced only within these sections of an SCL program.

CURROW is available in PROGRAM and FRAME entries.

Example

The following getrow section of a FRAME entry fetches a row from a previously opened SAS table, using _CURROW_ as the row number.

```
GET1:
   if fetchobs( tableid, _currow_ ) ne 0 then
      call notify('table','endtable');
return;
```

See Also

"_CURCOL_" on page 318

"SETROW" on page 664

CURSOR

Positions the cursor in a specified widget or field of a FRAME entry

Category: Widget or Field

Syntax

CURSOR *wvar-name*;

wvar-name
specifies which FRAME entry widget or field to position the cursor on.
Type: Character

Details The CURSOR statement does not move the cursor immediately while the SCL program is executing. Rather, it specifies where the cursor will be positioned when SCL returns control to the procedure after the frame is updated. If multiple cursor statements execute, the cursor is positioned on the FRAME entry widget or field that was specified in the last CURSOR statement. In SAS/AF applications, a REFRESH statement also positions the cursor, based on the last cursor statement.

The FRAME entry widget or field cannot be an element of an array. For an array element, use the FIELD function instead.

FRAME entry widgets can also use the _cursor method.

Example

Move the cursor to ADDRESS if NAME is filled in:

```
if modified (name) and name ne '' then
   cursor address;
```

See Also

"CURFLD" on page 319

"CURWORD" on page 324

"DISPLAYED" on page 353

"ERROR" on page 375

"ERROROFF" on page 376

"ERRORON" on page 377

"FIELD" on page 394

"HOME" on page 459

"MODIFIED" on page 550

"PROTECT" on page 602

"UNPROTECT" on page 695

CURTOP

Returns the number of the row that is currently displayed at the top of an extended table

Category: Extended Table

Syntax

row=**CURTOP**();

row
is the number of the row that is currently displayed at the top of an extended table.
Type: Numeric

Details CURTOP can be used only on extended tables in PROGRAM entries. Because extended tables can be defined only in SAS/AF software, this function cannot be used in FSEDIT or FSVIEW programs.

Example

Store the number of the table's top row in the column TOPROW:

```
toprow=curtop();
```

See Also

"ENDTABLE" on page 368

"SETROW" on page 664

"TOPROW" on page 689

CURWORD

Returns the word that is at the cursor position

Category: Widget or Field

Syntax

word=**CURWORD**();

word
is the text of the word.
Type: Character

Details CURWORD returns the word on which the text cursor was located when the user last pressed ENTER. The retrieved character string begins with the first character of the word on which the cursor was positioned and extends to the first space after the word. CURWORD is usually used in conjunction with a CONTROL statement that includes the ENTER, ALWAYS, or ALLCMDS option.

If CURWORD is used on a window variable that has been modified, the field value is justified before CURWORD executes.

Example

Suppose a PROGRAM entry window contains text entry widgets that contain the words **PROJECT1** and **PROJECT2**. The entry's program determines which action to perform by determining which word the cursor is positioned on when the user presses ENTER.

```
INIT:
   widget enter;
return;
MAIN:
   word=curword();
   if (word='PROJECT1') then
      submit continue;
         proc print data=project1;
         run;
      endsubmit;
   else if (word='PROJECT2') then
      submit continue;
         proc print data=project2;
         run;
      endsubmit;
   else _msg_=
     'Please position the cursor on a valid selection.';
return;

TERM:
return;
```

See Also

"CONTROL" on page 302

"CURFLD" on page 319

"CURSOR" on page 323

"LOOKUPC" on page 530

DATALISTC and DATALISTN

Displays a selection list window that contains the values of particular columns from rows in a SAS table and returns user selections

Category: SAS Table

Syntax

selection=**DATALISTC**(*table-id*,
  col-list<,*message*<,*autoclose*<,*num-sel*>>>);

selection=**DATALISTN**(*table-id*,
  col-list<,*message*<,*autoclose*<,*num-sel*>>>);

selection
is the value of the first column from the selected row. For DATALISTC, the value is the first character column. For DATALISTN, the value is the first numeric column. *Selection* is a missing value (period) if the user closes the selection list window without making a selection.

For DATALISTC, *selection* is blank if the selection list window is closed and no selections are made. By default, *selection* is 200 bytes long. To accommodate a value longer than 200 bytes, explicitly declare *selection* with a longer length.

Type: Character, Numeric.

table-id
is the identifier that was assigned when the table was opened. If *table-id* is invalid, the program halts.

Type: Numeric

col-list
is a list of column names, separated by blanks, from the SAS table to be displayed. For DATALISTC, the first column in this list must be character or else the program halts. For DATALISTN, the first column must be numeric or else the program halts. However, the remaining columns in the list can be of any type.

Type: Character

message
is the text for a message to be displayed above the selection list. The default message tells users to make up to the number of selections specified in *num-sel*.

Type: Character

autoclose
specifies whether the selection list window closes automatically after a user makes a selection when only one choice is allowed:

`'Y'`	closes the window automatically. (This is the default.)

'N' leaves the window open until the user explicitly closes it.

This option is ignored when *num-sel* is not 1. However, use `''`as a placeholder if you are also specifying a value for *num-sel*.

Type: Character

num-sel

specifies the maximum number of items a user can select from the list. To display the list for information purposes only (no selections allowed), specify **0**. To specify unlimited selections, use a value that is larger than the number of available selections, such as 9999.

Type: Numeric

Details If a user ends the selection list window without making a selection, DATALISTC and DATALISTN return a blank value. If the user exits by using $\boxed{\text{OK}}$ without making a selection, then the selection variable is set to blank. However, if the user exits using $\boxed{\text{CANCEL}}$ without making a selection, the selection variable retains any previous value it may have had.

Although a user can position the cursor or mouse pointer anywhere in a row to make a selection from the list, only the value of the first column is returned. (The other column values are displayed for information only.)

When multiple selections are allowed, *selection* contains only the value of the first column in the last selected row. However, values for displayed columns for all rows that are selected can be returned in the current result list if one is available. The current result list is a special SCL list that is automatically filled with the values selected from a selection list. To create a current result list, use the MAKELIST function to create it, and use the CURLIST function to designate it as the current result list. The current result list must exist before you call the DATALISTC or DATALISTN function.

By default, a message is displayed asking the user to make one selection, and the selection list window closes automatically when the user makes a selection.

When DATALISTC or DATALISTN is invoked, the current result list is cleared. After DATALISTC or DATALISTN is invoked, the result list contains the following named items:

TAG

identifies the list as one that was created by the DATALISTC function.

Type: Character

COUNT

contains either the number of selected elements, or 0 if a user makes no selections or issues a CANCEL command in the list window.

Type: Numeric

var-name

contains the value of column *var-name* in *var-list* for each selection.

Type: Numeric or Character

Examples

Example 1: Using DATALISTC to Return a Single Selection Create a selection list whose rows contain the values of the columns NAME, STREET, CITY, STATE, and ZIP from the SAS table identified by the SCL variable CLASSID, which was returned by the OPEN function. NAME contains the value for the selected row. The other columns are displayed for information purposes only.

```
name=datalistc(classid,'name street city state zip');
```

Example 2: Using DATALISTN to Return Multiple Selections Create a selection list whose rows contain the values of the columns ITEMNUM, ITEMAMT, CUSTNAM, and CUSTADR from the SAS table identified by the SCL variable SALESID. Allow users to make up to three selections from this selection list. Then retrieve the values for each column for each of the selected rows from the current result list.

```
salesid=open('sales');
listid=makelist();
rc=curlist(listid);
itemnum=datalistn(salesid,
  'itemnum itemamt custnam custadr','','',3);
n=getnitemn(listid,'COUNT');
do i=1 to n;
   itemnum=getnitemn(listid,'ITEMNUM',i);
   itemamt=getnitemn(listid,'ITEMAMT',i);
   custnam=getnitemc(listid,'CUSTNAM',i);
   custadr=getnitemc(listid,'CUSTADR',i);
   put itemnum= itemamt= custnam= custadr=;
end;
rc=close(salesid);
```

See Also

"LISTC and LISTN" on page 518

"LOCATEC and LOCATEN" on page 527

"SHOWLIST" on page 667

"VARLIST" on page 709

DCLOSE

Closes a directory

Category: Directory

Syntax

sysrc=**DCLOSE**(*directory-id*);

sysrc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

directory-id
is the identifier that was assigned when the directory was opened. If *directory-id* is invalid, the program halts.
Type: Numeric

Details The DCLOSE function closes a directory that was previously opened by the DOPEN function. DCLOSE also closes any open members of the directory before closing the directory.

Example

Open the directory to which the fileref MYDIR has previously been assigned, return the number of members, and then close the directory:

```
rc=filename('mydir','fname')
did=dopen('mydir');
memcount=dnum(did);
if (dclose(did)) then
   do;
      _msg_=sysmsg();
      ...SCL statements to handle the error condition...
   end;
```

See Also

"DOPEN" on page 357

"FCLOSE" on page 386

"FOPEN" on page 420

"MOPEN" on page 552

DCREATE

Creates an external directory

Category: External File

Syntax

new-directory=**DCREATE**(*dir-name*<,*parent-dir*>);

new-directory
contains the complete pathname of the new directory, or an empty string if the directory cannot be created.
Type: Character

dir-name
is the name of the directory to create. This must be only the directory name and cannot include a pathname.
Type: Character

parent-dir
is the complete pathname of the directory in which to create the new directory. If *parent-dir* is not supplied, then the current directory is the parent directory.
Type: Character

Details DCREATE enables you to create a directory in your operating environment.

Operating Environment Information: On CMS, DCREATE works only for shared file system (SFS) directories. △

Example

- □ Create a new directory on UNIX, using the name stored in the variable DIRNAME:

```
newdir=dcreate(dirname,'/local/u/abcdef/');
```

- □ Create a directory on Windows, using the name stored in the variable DIRNAME:

```
newdir=dcreate(dirname,'d:\testdir\');
```

- □ Create a new directory on CMS, using the name stored in the variable DIRNAME:

```
newdir=dcreate(dirname,'auser.');
```

DECLARE

Declares variables and specifies their data types

Alias: DCL

Category: Declarative Statement

Syntax

DECLARE|**DCL** *data-type-1 argument-1* < . . . ,*data-type-nargument-n* >;

data-type

specifies the data types to assign. Multiple data types may be declared with one DECLARE statement. Use a comma to separate multiple data types. If you omit a comma between data types, then subsequent data type names are interpreted as variable names until the next comma is encountered.

The following are valid data types:

'CHAR <**(***n***)**>'

is for variables that can contain character values. Optionally, *n* is the length. Default: 200. Declaring a length for CHAR variables that will store values shorter than 200 characters can reduce the amount of memory required to store a program's variables.

'LIST'

is for variables that can reference an SCL list.

'NUM'

is for variables that can contain numeric values.

'OBJECT'

is for variables that can contain the identifier of a component.

Note: The compiler cannot validate attributes or methods for objects declared with the OBJECT keyword (generic objects). Consequently, using generic objects is less efficient (possibly up to 25 percent less efficient) than declaring objects with the CLASS or INTERFACE keyword. See "Objects" on page 19 for more information. △

class-name
is for variables that can contain the identifier of an instance of a particular class. It can be a one– to four–level name.
Type: Character

argument-1< . . . argument-n>
can be either one or more constants and/or one or more variable names. The constants and/or variable names should be separated by spaces. Variable names can be any of the following:

variable
variable = initial-value
variable = expression
variable-1 – variable-n = (*value-1,...,value-n*)
listname={*value-1,...,value-n* }

Constants have the following form:

constant-n<=value-n>

Type: Character or Numeric (for variables).

Type: Character (for constants).

Details The DECLARE statement declares a variable of any SCL data type. DECLARE can be used within a DO, SELECT, or USECLASS block to define variables that are available only within that block. This enables you to enforce variable scoping, because variables that you declare within a DO, SELECT, or USECLASS block are local to that block.

You can use the DECLARE statement to declare any type of array. However, arrays that are declared with the DECLARE statement are all temporary arrays. See “Using Temporary Arrays to Conserve Memory” on page 46.

Although you can use the LENGTH statement to declare numeric and character variables, you might want to use the DECLARE statement in order to enforce variable scoping.

Place DECLARE statements either before the first labeled section of an SCL program or inside a DO or SELECT block.

Comparisons

- You can use the DECLARE statement to declare any SCL data type, whereas the LENGTH statement can declare only numeric and character variables.
- You can use the DECLARE statement inside a DO block, whereas the LENGTH statement cannot be used inside a DO block.
- You can use the DECLARE statement to declare temporary arrays, but you must use the ARRAY statement to declare indirect or non-temporary arrays.

For details about the LENGTH statement in the base SAS language, see *SAS Language Reference: Dictionary*.

Examples

```
dcl char  s;
dcl num x y;
dcl char s, num x y;
dcl char(10) ar[3] x y z;
dcl list mylist;
dcl sashelp.fsp.collection.class obj3;
dcl object obj4;
dcl num m n, char(300) string, list newlist;
```

Each variable or array can be followed by an initial value or expression. The following example declares and initializes various variables and arrays.

```
dcl num      x=1 y=20+x;
dcl num      i1-i4=(1, 2, 3, 4);
dcl num      arr(3)=(1, 2, 3);
dcl char(10) s='abc';
dcl char     sarr(3)=('abc', 'def', 'ghi');
dcl list     mylist = {1, 'abc', 2, 'def'};  /* Initialize a list */
dcl list l = (100, 'abc', 200);
```

Defining a Constant List with a Sublist

A constant list can be defined with a sublist.

```
init:            /*To edit a frame. As the frame runs,*/
                 /*it will display */
  dcl list l ;     /* a pop-up menu when you hit the ENTER key.*/;
  control enter
/* Initialize a list with three pop-menu items: Select 1, Numeric*/
/*and  Character. This will also define a separator between item*/
/*'Select 1'and 'Numeric'.   */
  list = { {text='Select 1',
              helpText='This is a selection.',
              mnemonic='S',
              classifier = 107},
         ''_'',
         ''Numeric'',
         ''Character''};

   return;
main:
 rc = popmenu (list);
 put rc=;
 return;

term:
 /* Delete the list recursively to avoid a memory leak */
 rc = dellist ( list, 'y');
 return;
```

Note: The form of the physical filename depends on the host operating system. △

See Also

"ARRAY" on page 254

"LENGTH" on page 510

"USECLASS" on page 698

DELARRAY

Deletes a dynamic array

Category: Array

Syntax

rc=**DELARRAY**(*array*);

rc
indicates whether the operation was successful.

0	successful
≠0	not successful

Type: Numeric

array
is the dynamic array to delete. A non-dynamic array causes an error condition.
Type: Array

Details The DELARRAY function deletes dynamic arrays. An array's contents cannot be accessed after the array is deleted. Dynamic arrays are only accessible within the scope that they are declared. In the following example, array B is not accessible outside of the scope of the DO group:

```
m1:method;
     DCL num a(*,*) rc;
     rc = makearray(a,5,5);
     do;
        DCL num b(*);
        rc = makearray(b,10);
     end;
endmethod;
```

Examples

Example 1: Create an Array and Delete It The example creates a 1-dimensional array of 5 elements, resizes it to 10 elements (preserving the data), and deletes the array.

```
DCL num a(*);
rc = makearray(a,3);
do i=1 to dim(a);
   a[i]=i;
end;
rc = redim(a,5);
put a=;
rc = delarray(a);
```

The output would be:

```
a[1]=1
a[2]=2
a[3]=3
a[4]=.
a[5]=.
```

See Also

“COPYARRAY” on page 309

“MAKEARRAY” on page 533

“REDIM” on page 610

DELETE

Deletes a member of a SAS data library or an external file or directory

Category: Utility

Syntax

sysrc=**DELETE**(*name*<,*type*,<,*password*<,*generation*>>>);

sysrc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

name
is the name of the member of the SAS data library or the physical pathname of an external file or directory. If a one-level name is specified for a SAS data library member, the library is assumed to be USER.
Type: Character

type
specifies the type of element to delete:

'ACCESS'
an access descriptor that was created using SAS/ACCESS software.

'CATALOG'
a SAS catalog or catalog entry. If a one- or two-level name is specified, the catalog is deleted. If a four-level name is specified, the entry is deleted.

'DATA'
a SAS table. (This is the default.)

'VIEW'
a SAS table view.

'FILE'
an external file or directory.
Type: Character

password
is the password that is assigned to the SAS table when *type* is **DATA**.
Type: Character

generation
is the generation number of the SAS table that is being deleted.
Type: Numeric

Details DELETE attempts to delete the specified member and returns a value indicating whether the operation was successful. You can use DELETE to delete files or empty directories that are external to a SAS session, as well as members of a SAS data library.

Examples

Example 1: Deleting Tables and Catalog Entries Delete the SAS table LIB1.MYDATA and the SAS catalog entry LIB2.CAT1.MYPROG.PROGRAM:

```
rc=delete('lib1.mydata');
rc=delete('lib2.cat1.myprog.program','catalog');
```

When deleting generation tables, if you delete the current (base) table without specifying the *generation* parameter, all tables in the generation group are deleted. For example:

```
rc=delete('one');
/* Deletes all tables in the generation group named 'one'*/
```

If you specify the current (base) table using the *generation* parameter, only that table is deleted. The youngest historical table becomes the new base. For example:

```
rc=delete('one','data','',0);
/* Deletes only the table work.one (relative generation number=0) */
```

Example 2: Deleting Files Delete an external file:

In UNIX:

```
/* delete a file in a different directory */
rc=delete('/local/u/abcdef/testfile','file');
```

In Windows:

```
/* delete a file in a different directory */
rc=delete('D:\testfile','file');
```

Example 3: Deleting Data Set Generations Delete the third generation of a data set:

```
   /* delete the third generation of the data set ‘work.one’ */
rc=delete('work.one','data','',3);
```

See Also

"NEW" on page 561

DELITEM

Deletes an item from an SCL list

Category: List

Syntax

list-id=**DELITEM**(*list-id*<,*index*>);

list-id
is the identifier of the list from which the item is to be deleted. The function returns the list identifier that is passed in. An invalid *list-id* produces an error condition.
Type: Numeric or List

index
is the position of the item in the list. The position can be specified as either a positive or negative number. By default, *index* is 1 (the first item). If *index* is a positive number, then the item is at position *index* from the beginning of the list. If *index* is a negative number, then the item is at position ABS(*index*) from the end of the list. An error condition results if the absolute value for *index* is zero or if it is greater than the number of items in the list.
Type: Numeric

Details The item to be deleted is specified by its position in the list that is passed in.

DELITEM does not make a copy of the list before deleting the specified item. The delete operation is performed in place on the list, and the list identifier is returned.

When the item to be deleted is a sublist, DELITEM deletes the item but not the sublist, because the sublist may be referenced by other SCL variables or lists.

An error condition results if

- the item has the NODELETE attribute
- the list has the NOUPDATE or FIXEDLENGTH attribute.

To check the attributes of a list or list item, use HASATTR. To change attributes, use SETLATTR.

See Also

"DELLIST" on page 336

"DELNITEM" on page 337

"POPC, POPL, POPN, and POPO" on page 593

DELLIST

Deletes a list and optionally deletes all of its sublists

Category: List

Syntax

rc=**DELLIST**(*list-id*<,*recursively*>);

rc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

list-id
is the identifier of the list to be deleted. An invalid *list-id* produces an error condition.
Type: Numeric or List

recursively

specifies whether to recursively delete all the list’s sublists and all sublists of its sublists.

`'N'`	Sublists are not deleted. (This is the default.)
`'Y'`	Sublists are deleted.

Type: Character

Details A list’s contents cannot be retrieved after the list is deleted.

If *recursively* is `'Y'`, DELLIST recursively deletes all sublists that do not have the NODELETE attribute. For sublists that have the NODELETE attribute, the sublist identifiers are removed from the deleted list, but the sublist is not deleted. Thus, you should store list identifiers for sublists either in another list or in an SCL variable so that you can access the lists later. All local lists that are not explicitly deleted are deleted when the application ends, at the same time that SCL closes open tables and files.

CAUTION:

Be careful when deleting lists recursively because you may inadvertently delete lists that are needed by other parts of the SCL program. Recursively deleting a list deletes all of its sublists even if they are referenced in other SCL lists or by other SCL variables. If you do not want a list to be deleted when it is a sublist item in a deleted list, use SETLATTR to assign the NODELETE attribute to the sublist. See “SETLATTR” on page 656 for a discussion of the NODELETE attribute. △

To conserve memory, delete lists when they are no longer needed. Typically, a DELLIST statement is placed in the termination section (TERM or FSETERM) of the program. Although the program that creates a list is most often responsible for deleting the lists that it creates, it does not have to delete them unless that is the appropriate action for the application; it may return the list that it created to its caller.

The list is not deleted, and a non-zero value is returned to *rc*, if

- the list has the NODELETE attribute
- the list is the local or global environment list (the lists returned by the ENVLIST function)
- *list-id* is a component identifier or a class list identifier.

To check attributes, use HASATTR. To change attributes, use SETLATTR.

If DELLIST fails because of a condition listed above, the list and/or sublists may be partially cleared, and no further items or sublists are cleared.

See Also

“CLEARLIST” on page 290

“DELITEM” on page 335

“DELNITEM” on page 337

“SETLATTR” on page 656

DELNITEM

Deletes a named item from an SCL list

Category: List

Syntax

list-id=**DELNITEM**(*list-id,name,<occurrence*
<,start-index<,index<,forceup>>>>);

list-id
is the identifier of the list from which the item is to be deleted. The function returns the list identifier that is passed in. An invalid *list-id* produces an error condition.
Type: Numeric or List

name
is the name of the item to delete. Item names are converted to uppercase and trailing blanks are ignored when searching the list for a matching name. Thus, the names 'abc' and 'Abc' are converted to 'ABC'.
Type: Character

occurrence
is the number of the occurrence of the named item to delete. The default, 1, specifies the first occurrence of the item.
Type: Numeric

start-index
specifies where in the list to begin searching for the item. By default, *start-index* is 1 (the first item). If *start-index* is positive, then the search begins at position *start-index* items from the beginning of the list. If *start-index* is negative, then the search begins at the item specified by ABS(*start-index*) items from the end of the list. An error condition results if the absolute value of *start-index* is zero or if it is greater than the number of items in the list.
Type: Numeric

index
specifies the variable to contain the position number of the deleted item. *Index* must be initialized to a nonmissing value; otherwise, errors result.

Note: This parameter is an update parameter. See "Input, Output, and Update Parameters" on page 34 for more information. △
Type: Numeric

forceup
can have one of the following values:

'Y'	specifies a case-insensitive search, which overrides the HONORCASE or NOHONORCASE list attribute.
'N'	specifies a search that uses the HONORCASE or NOHONORCASE list attribute and is the default action for lists when FORCEUP is not specified.
IGNORECASE	IGNORECASE is the alias for NOHONORCASE and is the default for a list. But you can use the SETLATTR function to set a list's attribute to HONORCASE.

Details DELNITEM searches for a named item and deletes it from the list. Case is ignored only if *forceup* is **'Y'**; otherwise, it searches according to the list attribute HONORCASE and NOHONORCASE.

If a list has the NOHONORCASE attribute, the case is also ignored.

If *occurrence* and *start-index* are both positive or both negative, then the search proceeds forward from the *start-index* item. For forward searches, the search continues only to the end of the list and does not wrap back to the front of the list. If *occurrence* or *start-index* is negative, then the search is backwards. For backward searches, the search continues only to the beginning of the list and does not wrap back to the end of the list.

DELNITEM does not make a copy of the list. The delete operation is performed in place on the list. For example, the following statement deletes the first item named **app** in the list identified by LISTID:

```
listid=delnitem(listid,'app');
```

When the item to be deleted is a sublist, DELNITEM deletes the item but not the sublist, because the sublist may be referenced by other SCL variables or lists.

An error condition results if

- the item has the NODELETE attribute
- the list has the NOUPDATE or FIXEDLENGTH attribute
- the named item is not found in the list.

To check the attributes of a list or list item, use HASATTR. To change these attributes, use SETLATTR.

Example

The following code creates a list named A, adds four items to the list, and prints the list. Then it deletes the third item in the list and prints the list again.

```
a=makelist();
rc=insertc(a,'a',-1,'var1');
rc=insertc(a,'b',-1,'var2');
rc=insertc(a,'c',-1,'var1');
rc=insertc(a,'d',-1,'var2');
call putlist(a,'Before deleting',0);
pos=0;
rc=delnitem(a,'var1',2,1,pos);
put pos=;
call putlist(a,'After deleting',0);
```

The results of this program are:

```
Before deleting(VAR1='a'
                VAR2='b'
                VAR1='c'
                VAR2='d'
                )[5]
POS=3
After deleting(VAR1='a'
               VAR2='b'
               VAR2='d'
               )[5]
```

Note: [5] is the list identifier that was assigned when this example was tested and may be different each time the example is run. △

See Also

"DELITEM" on page 335

"DELLIST" on page 336

"LISTLEN" on page 521

"SETLATTR" on page 656

DELOBS

Deletes a row from a SAS table

Category: SAS Table

Syntax

sysrc=**DELOBS**(*table-id*);

sysrc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

table-id
is the identifier that was assigned when the SAS table was opened. If *table-id* is invalid, the program halts.

Type: Numeric

Details You must fetch a row before it can be deleted. Some functions that fetch a row include FETCH, FETCHOBS, LOCATEC, LOCATEN, APPEND, DATALISTC, and DATALISTN.

Example

Delete the current row from an open SAS table. (The example assumes that the table identifier returned by the OPEN function was stored in the SCL variable MYDATAID.) If the function is unable to delete the row, a message is displayed on the message line.

```
if (delobs(mydataid)) then _msg_=sysmsg();
```

See Also

"APPEND" on page 251

"DATALISTC and DATALISTN" on page 326

"FETCH" on page 389

"FETCHOBS" on page 390

"LOCATEC and LOCATEN" on page 527

DESCRIBE

Fills an SCL list with items of system information about a SAS table, view, or catalog entry

Category: List

Syntax

sysrc=**DESCRIBE**(*source-name*,*list-id*<,*type*>);

sysrc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

source-name
is the name of the table, view, or catalog entry.
Type: Character

list-id
contains the identifier of an existing list to contain the control's description. An invalid *list-id* produces an error condition.
Type: Numeric or List

type
specifies the type of control to be described:

'CATALOG'
SAS catalog entry (This is the default.) For three- or four-level names, the default entry type is PROGRAM.

'DATA'
SAS table (This is the default for one- or two-level names.)

'VIEW'
SAS data view.
Type: Character

Details Because DESCRIBE replaces the previous values in the list, the same list can be used repeatedly.

The items of descriptive information are placed into the list as named items, and the names of the list items are the names of the attributes described. Only the named attributes that appear in the list are filled in.

The attributes that DESCRIBE can place in an SCL list are determined by the value of *type*. If *type* is DATA or VIEW, then the items named in the list are attributes that are returned by the ATTRN and ATTRC functions. If *type* is CATALOG, then the items named in the list are DESC (the description of the catalog), EDESC (the extended description of the catalog entry), CRDATE (the date that the entry was created), and DATE (the date that the entry was last modified). CRDATE and DATE are SAS date values.

For catalog entries, if a numeric list item named DATE exists in the list, then DESCRIBE sets that item's values to a SAS date value. Otherwise, if DATE is a character list item, then DESCRIBE assigns a formatted date string using the MMDDYY10. format. Use ITEMTYPE to determine the type of a list item.

Example

Create an SCL list containing items named DESC and DATE. DESCRIBE fills the DESC and DATE items in the list with information about the catalog entry MYLIB.MYCAT.A.SCL.

```
init:
  desc_list=makenlist('L','DESC','DATE');
 /* set DATE to character list item */
  rc=setnitemc(desc_list,' ','DATE');
  rc=describe('MYLIB.MYCAT.A.SCL',desc_list,'CATALOG');
  call putlist(desc_list);
return;
```

The output would be similar to:

```
(DESC='A.SCL' DATE='03/25/1999' )[5]
```

See Also

"ATTRC and ATTRN" on page 258

DEVLIST

Displays a selection list of graphic hardware devices and returns user selections

Category: Selection List

Syntax

selections=**DEVLIST**(*catalog-name,device*<,*message*
<,*autoclose*<,*num-sel*>>>);

selections
are one or more user selections from the list, or blank if the selection list window is closed and no selections are made.
Type: Character

catalog-name
is the catalog that lists the devices. Usually the catalog SASHELP.DEVICES is used as *catalog-name*.
Type: Character

device
is the type of device to be listed:

'CAMERA'
lists device catalog entries for film recorders.

'DEFAULT'
returns the current device name, description, and type instead of displaying a list.

'EXPORT'
lists device catalog entries for device drivers that produce a graphics stream file.

'MONITOR'
lists device catalog entries for video displays.

'PLOTTER'
lists device catalog entries for plotters.

'PRINTER'
lists device catalog entries for printers.
Type: Character

message
is the text for a message to be displayed above the selection list. The default message tells users to make up to the number of selections specified in *num-sel*.
Type: Character

autoclose
specifies whether the selection list window closes automatically after a user makes a selection when only one choice is allowed:

'Y'	closes the window automatically. (This is the default.)
'N'	leaves the window open until the user explicitly closes it.

This option is ignored when *num-sel* is not 1. However, use `''`as a placeholder if you are also specifying a value for *num-sel*.
Type: Character

num-sel
specifies the maximum number of items a user can select from the list. To display the list for information purposes only (no selections allowed), specify 0. To specify an unlimited number of selections, use a value that is larger than the number of available selections about graphic device drivers such as 9999.
Type: Numeric

Details The value in *selections* consists of a 40-character description, an 8-character device name, and an 8-character device type. For additional details about graphic device drivers, see *SAS/GRAPH Software: Reference* and the SAS online Help for SAS/GRAPH software.

You can provide a default value or an initial selected value in the list by providing a value for the *selections* variable before calling DEVLIST. If *selections* contains valid entry names when the function is invoked, those names are automatically designated as selected when the selection list is displayed.

If a user closes the selection list window without making a selection, then DEVLIST returns a blank value unless there was an initial value for the *selections* variable before DEVLIST was called.

When multiple selections are allowed, *selections* contains the first value selected from the list. However, the values for all selections can be returned in the current result list, if one is available. The current result list is a special SCL list that is automatically filled with the values selected from a selection list. To use a current result list, use the

MAKELIST function to create the list, and use the CURLIST function to designate it as the current result list. The current result list must exist before you call DEVLIST.

When DEVLIST is invoked, the current result list is cleared. After DEVLIST is invoked, the result list contains the following named items:

TAG
: identifies the list as one that was created by the DEVLIST function.
  Type: Character

COUNT
: contains either the number of selected elements, or 0 if a user makes no selections or issues a CANCEL command in the list window.
  Type: Numeric

DESC
: contains the description of the selected device. There is one DESC element for each selection. The value of DESC is in the case that was entered originally.
  Type: Character

DEVICE
: contains the name for each selected device. There is one DEVICE element for each selection.
  Type: Character

TYPE
: contains the type of each selected device. There is one TYPE element for each selection.
  Type: Character

Examples

Example 1: Displaying the Printer Devices of a Catalog Display a list of devices of type PRINTER that are available in the catalog SASHELP.DEVICES. After the user selects one device from the list, the program uses the SUBSTRING function to extract the individual items of information returned by DEVLIST.

```
select=devlist('sashelp.devices','printer',
  'Select a device.');
descript=substr(select,1,40);
device=substr(select,41,8);
devtype=substr(select,49,8);
```

Example 2: Using the Results of a DEVLIST Use the current result list to process multiple selections:

```
listid=makelist();
rc=curlist(listid);
selection=devlist('sashelp.devices','printer',
  'Select a device',' ',3);
n=getnitemn(listid,'COUNT');
do i=1 to n;
   descript=getnitemc(listid,'DESC',i);
   device=getnitemc(listid,'DEVICE',i);
   devtype=getnitemc(listid,'TYPE',i);
   put descript= device= devtype=;
end;
```

See Also

"COLORLIST" on page 293

DIALOG

Runs a FRAME entry that was created with SAS/AF software and disables all other windows

Category: Modular Programming

Syntax

CALL DIALOG(*entry*<,*parameters*>)

entry
is a FRAME entry to be displayed. It is specified as

entry.type
for a FRAME entry in the current catalog.

libref.catalog.entry.type
for a FRAME entry in a specified catalog.
Type: Character

parameters
lists one or more parameters to pass to the called entry. In order for the called entry to accept these parameters, it must contain a corresponding ENTRY statement.

Note: These parameters are update parameters. See "Input, Output, and Update Parameters" on page 34 for more information. △
Type: Numeric, Character

Details

DIALOG runs a FRAME entry, makes it the active entry, and disables all other windows. When the called entry is exited, control returns to the calling program. With the exception of disabling all other windows, DIALOG is similar to DISPLAY.

Note: From the window created with the CALL DIALOG routine, you cannot execute a SUBMIT statement with the CONTINUE option. See "Controlling What Happens After a Submit Block Executes" on page 83. △

DIALOG can pass parameters through the ENTRY statement to the called Frame entry. Parameters can be numeric constants, character constants, variables, expressions, and array variables.

Using DIALOG without any options in the associated ENTRY statement requires a strict correspondence between DIALOG parameters and ENTRY statement arguments. The arguments and parameters must agree in number, data type, and relative position. If you pass an incorrect number of parameters or a parameter of the incorrect type, SCL halts the execution of the program. The argument-parameter correspondence is less restrictive when you use the options REST=, ARGLIST=, and OPTIONAL= in the ENTRY statement.

Names listed in *parameter* do not have to match the argument names in the ENTRY statement.

Parameters are passed in the following ways:

call-by-reference
: passes window variables and local variables and allows values to be returned to the calling program. This method allows the called program to modify values and then return them. If you do not want to return the new values, use the NOCHANGE routine. Or, if you do not want to return the new values for particular parameters, use the INPUT option for that parameter in the ENTRY statement. Here is an example of call-by-reference:

```
array employee{50};
call dialog('b.frame',var1,name,num,employee{1});
```

call-by-value
: is used for all numeric constants, character constants, and expressions. It does not allow values to be returned to the calling program. Here is an example of call-by-value:

```
call dialog('b.frame',100,'hello',x+y);
```

See Also

"DISPLAY" on page 350

"ENTRY" on page 369

"NOCHANGE" on page 569

"RETURN" on page 615

DINFO

Returns information about a directory

Category: Directory

Syntax

attribute=**DINFO**(*directory-id*,*info-item*);

attribute
: contains the value of the information item, or a blank if *info-item* is invalid.
  Type: Character

directory-id
: is the identifier that was assigned when the directory was opened. If *directory-id* is invalid, the program halts.
  Type: Numeric

info-item
: is the information item to be retrieved.
  Type: Character

Details DINFO returns the value of a system-dependent directory parameter. The available information varies according to the operating system. See the SAS documentation for your host operating system for information about system-dependent directory parameters.

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.

Example

Open the directory MYDIR, determine the number of directory information items available, and retrieve the value of the last item:

```
   /* Assign the fileref MYDIR to the */
   /* pathname stored in the variable */
   /* or entered in the DIRNAME field and open it.*/
   rc=filename('mydir',dirname);
did=dopen('mydir');
numopts=doptnum(did);
foption=doptname(did,numopts);
charval=dinfo(did,foption);
rc=dclose(did);
```

See Also

"DOPTNAME" on page 358

"DOPTNUM" on page 359

"FINFO" on page 412

"FOPTNAME" on page 421

"FOPTNUM" on page 423

DIRLIST

Opens a host selection list window that lists members of one or more SAS data libraries, and returns a user's selections

Category: Selection List

Syntax

selections=**DIRLIST**(*lib-spec*,*member-type*,*num-sel*,*prefix*<,*table-type*<,*sel-excl*<,*message*>>>);

selections
contains one or more user selections. Multiple selections are separated by blanks. By default, *selections* is 200 bytes long. To accommodate values longer than 200 bytes, explicitly declare *selections* with a longer length.
Type: Character

lib-spec
lists one or more librefs that are associated with particular SAS data libraries. To include or exclude SAS data libraries, use a name specification style from "Name Specifications for Arguments" on page 348. By default, SASHELP is not included in the selection window.
Type: Character

member-type
lists one or more types of SAS data library members. For example, a few common *member-types* are DATA, VIEW, and CATALOG. To include or exclude particular *member-types*, use a name specification style from "Name Specifications for Arguments" on page 348.
Type: Character

num-sel
is the maximum number of items that a user can select from the list. To display the list for information purposes only (no selections allowed), specify 0. To specify an unlimited number of selections, use a value such as 9999 that is larger than the number of available selections.
Type: Numeric

prefix
specifies whether names that are selected are prefixed with the libref:

'Y'
Selected names are returned as *libref.name*.

'N' or **' '**
Selected names are returned as *name*.

table-type
lists one or more SAS table types. By default, the selection list displays members of all SAS table types. To include or exclude specific table types, use a name specification style from "Name Specifications for Arguments" on page 348. This argument is ignored unless **DATA** is one of the values of *member-type*. For information about table types, see the description of the TYPE= data set option in *SAS Language Reference: Dictionary*.
Type: Character

sel-excl
lists one or more SAS data library members to include or exclude from the list. Use a name specification style from "Name Specifications for Arguments" on page 348. If *prefix* is **N**, then specify the name here as *member*. If *prefix* is **Y**, then specify the name here as *libref.member*.
Type: Character

message
is the text for a message to be displayed above the selection list. The default message tells users to make up to the number of selections specified in *num-sel*.
Type: Character

autoclose
is an obsolete arument bur is retained for compatibility with earlier releases. If you want to specify a value for *num-sel*, then specify `''` as a placeholder for this argument.

Name Specifications for Arguments

For *lib-spec*, *member-type*, and *table-type*, use these guidelines for specifying names:

□ To specify one or more specific names, separate the names with a space.
□ To specify all names, use an asterisk (`*`) or a null string (`''`).
□ To specify all names *except* those listed, use a NOT sign (^ or ¬) followed by one or more names.

Details If a user closes the selection list window without making a selection, *selections* contains a blank value unless that variable contained a valid value before DIRLIST was called.

The values for all selections can be returned in the current result list, if one is available. The current result list is a special SCL list that is automatically filled with the values selected from a selection list. To use a current result list, use the MAKELIST function to create the list, and use the CURLIST function to designate it as the current result list. The current result list must exist before you call the DIRLIST function.

When the function is invoked, the current result list is cleared. After DIRLIST is invoked, the current result list contains the following named elements:

TAG
: identifies the list as one that was created by DIRLIST.
  Type: Character

COUNT
: contains the number of selected items or contains 0 if a user makes no selections or issues a CANCEL command in the list window.
  Type: Numeric

NAME
: contains the uppercase name of each selected catalog entry. If *prefix* is `Y`, then the name is in the form *libref.member*. Otherwise, it is in the form *member*. There is one NAME element for each selection made.
  Type: Character

DESC
: contains the description of each selected catalog entry. There is one DESC element for each selection made. The value of DESC is in the case entered originally. If the SAS system option DETAILS is in effect, then DESC contains the table label.
  Type: Character

TYPE
: contains the type of each selected library member. There is one TYPE element for each selection.
  Type: Character

Because some engines support mixed-case filenames, DIRLIST now retains the cases of the returned selected items. This may cause your application to fail if your application contains code that assumes the returned selection is uppercased. For example,

```
if (dirlist(dsid, 'TESTNDX')='NDXVAR')
```

must be changed to

```
if (upcase(dirlist(dsid, 'TESTNDX'))='NDXVAR'
```

If the application cannot be modified, you may need to specify the VALIDVARNAME=V6 system option when you run the application to ensure that the selections returned from the DIRLIST function will be uppercased.

Example

Display a selection list of SAS tables in the SAS libraries MYLIB1 and MYLIB2 except MYLIB1.ANSWERS, and enable users to select up to three table names. The selections are retrieved from the current environment list by using GETNITEMC.

```
listid=makelist();
rc=curlist(listid);
selections=dirlist('mylib1 mylib2','data',3,'Y',
                   ' ','^ mylib1.answers');
n=getnitemn(listid,'COUNT');
do i=1 to n;
```

```
      member=getnitemc(listid,'NAME',i);
      descript=getnitemc(listid,'DESC',i);
      memtype=getnitemc(listid,'TYPE',i);
      put member= descript= memtype=;
   end;
```

See Also

"CATLIST" on page 269

"FILELIST" on page 401

"LIBLIST" on page 513

DISPLAY

Runs a catalog entry that was created with SAS/AF software

Category: Modular Programming and Object Oriented

Syntax

CALL DISPLAY(*entry*<,*parameters*>);

return-value=**DISPLAY**(*entry*<,*parameters*>);

entry
is a display entry (FRAME, PROGRAM, SCL, MENU, HELP, or CBT) that was created using SAS/AF software. It is specified as

entry-name
for a PROGRAM entry in the current catalog.

entry.type
for an entry of the specified type in the current catalog.

libref.catalog.entry
for a PROGRAM entry in the specified catalog.

libref.catalog.entry.type
for an entry of a specified type in a specified catalog.
Type: Character

parameters
lists one or more parameters to pass to the called entry. You can pass parameters to FRAME, PROGRAM, and SCL entries. In order for the called entry to accept these parameters, it must contain a corresponding ENTRY statement.

Note: These parameters are update parameters. See "Input, Output, and Update Parameters" on page 34 for more information. △
Type: Numeric, Character

return-value
contains the value that is returned by the called entry. The data type for *return-value* should match the data type for the called entry.
Type: Numeric, Character, List, Object, Class, or Interface

Details DISPLAY can run a FRAME, PROGRAM, SCL, MENU, HELP, or CBT entry and make it the active entry. When the called entry is exited, control returns to the calling program.

DISPLAY can pass parameters to a FRAME, PROGRAM, or SCL entry and receive a return value. Parameters can be numeric constants, character constants, variables, expressions, and array variables. Parameters are passed to the ENTRY statement in the called entry.

Using DISPLAY without any options in the associated ENTRY statement requires a strict correspondence between DISPLAY parameters and ENTRY statement arguments. The arguments and parameters must agree in number, data type, and relative position. If you pass an incorrect number of parameters or a parameter of the incorrect type, SCL halts the execution of the program. The argument-parameter correspondence is less restrictive when you use the options REST=, ARGLIST=, and OPTIONAL= in the ENTRY statement. See “ENTRY” on page 369 for examples of these options.

Names listed in *parameter* do not have to match the argument names in the ENTRY statement.

Parameters are passed in the following ways:

call-by-reference
: enables the specified entry to change the values of the parameters. If you do not want the values of the parameters to be modified, use the NOCHANGE routine. Or, if you do not want to return the new values for specific parameters, use the INPUT option for that parameter in the ENTRY statement. Here is an example of call-by-reference:

```
array employee{50};
call display('b.frame',var1,name,num,employee{1});
```

call-by-value
: prevents the specified entry from changing the values of the parameters. Call-by-value is used for all numeric constants, character constants, and expressions. Here is an example of call-by-value:

```
call display('b.frame',100,'hello',x+y);
```

Note: Use CALL CBT to run CBT applications, because it provides more options used by CBT entries. In general, you may want to use CALL GOTO instead of CALL DISPLAY if you do not need control to return to the calling program. This may be helpful for applications that have memory constraints. △

Examples

Example 1: Passing Parameters Use DISPLAY in program X to pass parameters to program Y. Program Y then declares these arguments with an ENTRY statement. Variables I and S are call-by-reference parameters, and the constant 1 is a call-by-value parameter.

X.SCL contains the following program:

```
INIT:
   s = 'abcd';
   i = 2;
   call display('y.frame', i, 1, s);
      /* At this point, after the return from Y, */
      /* i=7 and s='abcde'                        */
   put i= s=;
```

```
return;
MAIN:
TERM:
return;
```

Y.SCL contains the following program:

```
entry j c:num str:char;
init:
   j = length(str) + c;
   j = j + 2;
   str = str || 'e';
   c = 2;
return;
```

The following correspondence occurs:

- □ The value of variable I passes to variable J.
- □ The literal value 1 passes to variable C.
- □ The value of variable S passes to variable STR.

After program Y runs, the values of variables J and STR are returned to the variables I and S, respectively. The variable C cannot return a value, however, because the corresponding parameter in DISPLAY is a constant.

Example 2: Passing Array Parameters by Reference Use DISPLAY to pass array parameters by reference. In this example, the variables S and A are call-by-reference parameters, and the constant 4 is a call-by-value parameter.

X.SCL contains the following program:

```
array a{4} 8;
INIT:
   a{1} = 1; a{2} = 2; a{3} = 3; a{4} = 4;
   s = 0;
   call display('y.frame', s, a, 4);
     /*  At this point, after the return
                              from Y, */
     /*  s=10, a{1}=2, a{2}=4, a{3}=6,
                            a{4}=8.   */
   put s= a=;
return;

MAIN:
TERM:
return;
```

Y.SCL contains the following program:

```
array arr{*} 8;
entry sum arr[*] len:num;
INIT:
   do i = 1 to len;
      sum = sum + arr{i};
      arr{i} = 2 * arr{i};
   end;
return;
```

The following correspondence occurs:

- □ The value of variable S passes to variable SUM.
- □ The array variable A is passed to the array variable ARR.
- □ The literal value 4 passes to variable LEN.

After program Y runs, the value of the variable SUM is returned to the variable S, and the values in the array ARR are returned to the corresponding values in the array A. The variable LEN cannot return a value, however, because the corresponding parameter in DISPLAY is a constant.

See Also

"DIALOG" on page 345

"ENTRY" on page 369

"GOTO" on page 455

"INPUTC and INPUTN" on page 481

"NOCHANGE" on page 569

"RETURN" on page 615

DISPLAYED

Reports whether a control or field is currently visible

Category: Control or Field

Syntax

rc=**DISPLAYED**(*wvar-name*);

rc
indicates whether the FRAME entry control or field is visible:

1	visible
0	not visible

Type: Numeric

wvar-name
is the name of a control or field. This name cannot be an element of an array or an expression. An invalid *wvar-name* halts the program.

Type: Character

Details In SAS/AF applications, DISPLAYED reports whether a control is currently visible or whether it has the NONDISPLAY attribute. In FSEDIT, DISPLAYED tells you whether a field is visible on the current screen of a multiscreen application, regardless of whether the control has the NONDISPLAY attribute. This function is useful in multiscreen applications in which the application developer wants to be on a specific screen for a field.

If a control is not currently displayed, then your application can use the EXECCMD routine to issue scrolling commands to change the screen position or to issue scrolling commands that are specific to the procedure (for example, the =*n* command in the FSEDIT procedure).

The control or field cannot be an element of an array. To report this information for an array element, use FIELD instead.

FRAME entry controls can also use the _isDisplayed or _isHidden method.

Example

Test whether the SALARY field is displayed on the current screen of an FSEDIT application. If not, issue an FSEDIT scrolling command to display the screen that contains the field:

```
if (displayed(salary)=0) then
   call execcmd('=salary');
```

See Also

"ERROR" on page 375

"ERROROFF" on page 376

"ERRORON" on page 377

"FIELD" on page 394

"MODIFIED" on page 550

DMWINDOW

Sets the color and highlighting for lines in the OUTPUT and LOG windows

Category: Window

Syntax

rc=**DMWINDOW**(*window-name*,*line-type*,*color*,*attribute*);

rc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

window-name
is the window for which you want to assign colors and display attributes:

'OUTPUT'
the OUTPUT window

'LOG'
the LOG window
Type: Character

line-type
is the output area to which the colors and highlighting attributes are to be assigned:

'DATA'
Data line

'ERROR'
Error line

'NOTES'
Notes line (LOG window only)

'SOURCE'
Source line (LOG window only)

'WARNING'
Warning line (LOG window only)

'BYLINE'
Byline line (OUTPUT window only)

'HEADER'
Header line (OUTPUT window only)

'TITLE'
Title line (OUTPUT window only)
Type: Character

color
is a color name: BLACK, BLUE, BROWN, CYAN, GRAY, GREEN, MAGENTA, ORANGE, PINK, RED, WHITE, or YELLOW. SASCOLOR window elements can also be used for *color*.
Type: Character

attribute
is a display attribute: NONE, BLINKING, HIGHLIGHT, HIREV, REVERSE, or UNDERLINE. If you specify a SASCOLOR window element for *color*, then *attribute* is ignored, because the SASCOLOR window element contains a display attribute.
Type: Character

Details The device must support the specified color or highlighting attribute in order for SAS to enable the attribute.

Example

Set the highlighting attribute of the title line in the OUTPUT window to blinking and set its color to yellow:

```
rc=dmwindow('output','title','yellow','blinking');
```

DNUM

Returns the number of members in a directory

Category: Directory

Syntax

nval=**DNUM**(*directory-id*);

nval
contains the number of members in the directory.
Type: Numeric

directory-id
is the identifier that was assigned when the directory was opened. If *directory-id* is invalid, the program halts.
Type: Numeric

Details You can use DNUM to determine the largest member number that can be passed to DREAD.

Example

Open the directory MYDIR, determine the number of members, and close the directory:

```
     /* Assign the fileref MYDIR to the          */
     /* filename stored in the variable DIRNAME  */
     /* and open it.                             */
  rc=filename('mydir',dirname);
  dirid=dopen('mydir');
  memcount=dnum(dirid);
  rc=dclose(dirid);
```

See Also

"DREAD" on page 360

DO

Designates a group of statements to be executed as a unit

Category: Control Flow

Comparisons: SAS Statement with limitations in SCL

Syntax

DO *do-clause*;

END;

Details The following forms of the *do-clause* are supported:

iterative DO
executes a group of statements repetitively, based on the value of an index variable. However, the form DO *i=item-1, . . . , item-n* is not supported.

DO UNTIL
executes a group of statements repetitively until a condition is true.

DO WHILE
executes a group of statements repetitively as long as a condition is true.

The form DO OVER is not supported.

To force the statements in a DO group to stop executing, you can use the SCL statements CONTINUE or LEAVE.

For details about the DO statement in the base SAS language, see *SAS Language Reference: Dictionary*.

See Also

"CONTINUE" on page 300

"LEAVE" on page 504

DOPEN

Opens a directory

Category: Directory

Syntax

directory-id=**DOPEN**(*fileref*);

directory-id
contains the return code for the operation:

0	indicates that the directory could not be opened.
>0	is the identifier that was assigned to the opened directory.

Type: Numeric

fileref
is the fileref that is assigned to the directory.
Type: Character

Details DOPEN opens a directory and returns a directory identifier value (a number greater than 0), which can then be used to identify the open directory to other SCL functions. The directory to be opened must be identified by a fileref. You must associate a fileref with the directory before calling DOPEN.

You can assign filerefs by using either the FILENAME statement or the FILENAME function in SCL. Under some operating systems, you can also use system commands to assign filerefs.

Operating Environment Information: The term *directory* used in the description of this function and related SCL functions refers to an aggregate grouping of files that are managed by the host operating system. Different host operating systems identify such groupings with different names, such as directory, subdirectory, MACLIB, or partitioned data set. See the SAS documentation for your operating environment for details. △

Example

Assign the fileref MYDIR to a directory. Then open the directory, determine how many system-dependent directory information items are available, and close the directory. DIRNAME is an SCL variable with a value that represents the actual name of the directory in the form required by the host operating system.

```
rc=filename('mydir',dirname);
did=dopen('mydir');
infocnt=doptnum(did);
rc=dclose(did);
```

See Also

"DCLOSE" on page 328

"FOPEN" on page 420

"MOPEN" on page 552

DOPTNAME

Returns the name of a directory attribute

Category: Directory

Syntax

attribute=**DOPTNAME**(*directory-id*,*attribute-number*);

attribute
contains the directory option. If *nval* is out-of-range, the program halts and *attribute* contains the value that it held before the program halt.
Type: Character

directory-id
contains the identifier that was assigned when the directory was opened. If *directory-id* is invalid, the program halts.
Type: Numeric

attribute-number
is the sequence number of the option.
Type: Numeric

Details DOPTNAME works only if the directory was previously opened by the DOPEN function. The names and nature of directory information items vary depending on the operating system. The number of attributes that are available for a directory also varies depending on the operating system.

Example

Open the directory identified by the fileref MYDIR, retrieve all system-dependent directory information items, write them to the SAS log, and then close the directory:

```
   /* Assign the fileref MYDIR to the  */
   /* filename stored in the variable DIRNAME   */
   /* and open it.                              */
rc=filename('mydir',dirname);
dirid=dopen('mydir');
numitems=doptnum(dirid);
do j=1 to numitems;
   opt=doptname(dirid,j);
   put 'Directory information=' opt;
end;
rc=dclose(dirid);
```

See Also

"DINFO" on page 346

"DOPTNUM" on page 359

DOPTNUM

Returns the number of information items that are available for a directory

Category: Directory

Syntax

num-attributes=**DOPTNUM**(*directory-id*);

num-attributes
contains the number of available directory information items. If an error condition occurs, the program halts and *num-attributes* contains the value that it held before the program halt.
Type: Numeric

directory-id
contains the identifier that was assigned when the directory was opened. If *directory-id* is invalid, the program halts.
Type: Numeric

Details DOPTNUM works only if the directory was previously opened by the DOPEN function.

Example

Retrieve the number of system-dependent directory information items that are available for the directory MYDIR and then close the directory:

```
   /* Assign the fileref MYDIR to the           */
   /* filename stored in the variable DIRNAME   */
   /* and open it.                              */
rc=filename('mydir',dirname);
dirid=dopen('mydir');
infocnt=doptnum(dirid);
rc=dclose(dirid);
```

See Also

"DINFO" on page 346

"DOPTNAME" on page 358

DREAD

Returns the name of a directory member

Category: Directory

Syntax

name=**DREAD**(*directory-id*,*member-num*);

name
contains either the name of the member, or a blank if an error occurs (for example, if *nval* is out-of-range).
Type: Character

directory-id
contains the identifier that was assigned when the directory was opened. If *directory-id* is invalid, the program halts.
Type: Numeric

member-num
is the sequence number of the member within the directory.
Type: Numeric

Details Use DNUM to determine the highest possible member number that can be passed to DREAD. DREAD works only if the directory was previously opened by the DOPEN function.

Example

Open the directory identified by the fileref MYDIR, retrieve the number of members and place the number in the variable MEMCOUNT, retrieve the name of the last member and place the name in the variable LSTNAME, and then close the directory:

```
   /* Assign the fileref MYDIR to the          */
   /* filename stored in the variable DIRNAME  */
   /* and open it.                             */
rc=filename('mydir',dirname);
dirid=dopen('mydir')
lstname='';
memcount=dnum(dirid);
if (memcount>0) then
   lstname=dread(dirid,memcount);
rc=dclose(dirid);
```

See Also

"DNUM" on page 355

"DOPEN" on page 357

DROPNOTE

Deletes a note marker from either a SAS table or an external file

Category: SAS Table

Syntax

rc=**DROPNOTE**(*table-id* | *file-id*,*note-id*);

rc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

table-id **or** ***file-id***
contains the identifier that was assigned when the table or external file was opened. If this variable contains an invalid value, the program halts.
Type: Numeric

note-id
contains the identifier that was assigned by the NOTE or FNOTE function. If *note-id* contains an invalid value, the program halts.
Type: Numeric

Details DROPNOTE deletes a marker that was set by NOTE or FNOTE.

Example

Open the SAS table MYDATA, fetch the first row, and set a note ID at the beginning of the table. Return to the first row by calling POINT, and then delete the note ID by calling DROPNOTE.

```
dsid=open('mydata','i');
rc=fetch(dsid);
noteid=note(dsid);
   /* more SCL statements */
rc=point(dsid,noteid);
rc=fetch(dsid);
rc=dropnote(dsid,noteid);
```

See Also

"FNOTE" on page 417

"FPOINT" on page 425

"NOTE" on page 570

"POINT" on page 592

DSID

Searches for a SAS table name and returns the table identifier

Category: SAS Table

Syntax

dsid=**DSID**(<*table-name*<,*mode*<,*nth*<,*gen-num*>>>>);

dsid
contains either the identifier for the table, or

0	if the table is not currently open, if the table is not open in the requested mode, or if no *nth* open occurrence exists.
<0	if an error occurs. SYSMSG contains the error text.

Type: Numeric

table-name
names the SAS table to search for. The default is _LAST_, which is the last table that was created in the current SAS session. A one-level name is assumed to be a SAS table name in the default SAS data library, WORK. A two-level name is assumed to be *libref.table*.

Type: Character

mode
specifies whether to limit the search to tables that are open in one of the modes listed below. If *mode* is not specified, DSID returns the *dsid* for the first occurrence of *table-name* that is open in any mode. Values for *mode* are

'I'	INPUT mode, allows random access if the engine supports it; otherwise, defaults to IN mode.
'IN'	INPUT mode, reads sequentially and allows revisiting rows.
'IS'	INPUT mode, reads sequentially but does not allow revisiting rows.
'N'	NEW mode, creates a new SAS table.
'U'	UPDATE mode, allows random access if the engine supports it; otherwise, defaults to UN mode.
'UN'	UPDATE mode, reads sequentially and allows revisiting rows.
'US'	UPDATE mode, reads sequentially but does not allow revisiting rows.

'V'	UTILITY mode, allows modification of column attributes and indexes that are associated with the SAS table.

For more information about open modes, see "OPEN" on page 576.
Type: Character

nth
specifies which occurrence of *table-name* opened in the specified *mode* to search for. By default, the search returns the first occurrence.
Type: Numeric

gen-num
is the generation number of the SAS table for which the DSID is returned.
Type: Numeric

Details DSID searches all SAS tables that are currently open. This function is useful for accessing table identifiers across entries.

Examples

Example 1: Working with Several Tables Open several SAS tables and find the first occurrence in various modes:

```
/* Open several SAS tables, varying the open mode */
dsid1 = open('sasuser.class', 'I');
dsid2 = open('sasuser.class', 'U');
dsid3 = open('sasuser.class', 'U');
dsid4 = open('sasuser.houses', 'U');
dsid5 = open('sasuser.class', 'I');
dsid6 = open('sasuser.houses', 'U');
dsid7 = open('sasuser.houses', 'I');
dsid8 = open('sasuser.class', 'U');

 /* Find the first occurrence open in any mode.*/
first = DSID( 'sasuser.houses' );
put first=;

 /* Find the first occurrence open in 'I' */
firstI = DSID( 'sasuser.houses', 'I' );
put firstI=;

 /* Find the second occurrence open in 'I' */
secondI = DSID( 'sasuser.class', 'I', 2 );
put second=;

 /* Return the fourth occurrence open in 'U' */
secondU = DSID( 'sasuser.class', 'U', 4 );
put secondU=;
```

This example produces the following output:

```
first=4
firstI=7
secondI=5
secondU=0
```

Example 2: Returning the DSID of a Generation Data Set The following code returns the DSID of the SAS table WORK.ONE#003.

```
dsid=DSID('work.one','IN',1,3);
```

See Also

"OPEN" on page 576

DSNAME

Returns the SAS table name that is associated with a table identifier

Category: SAS Table

Syntax

table-name=**DSNAME**(<*table-id*>);

table-name
contains either the table name that is associated with the specified *table-id* value, or a blank if an invalid value is specified.
Type: Character

table-id
contains the identifier that was assigned when the table was opened. If DSNAME is called from FSEDIT, FSBROWSE, or FSVIEW, this value is optional. If *table-id* is not specified in an FSEDIT or FSVIEW application, then DSNAME returns the name of the current table. If *table-id* is not specified in a SAS/AF application, then DSNAME returns a blank.
Type: Character

Examples

- Determine the name of the SAS table that is associated with the table identifier TABLEID and display this name on the message line:

```
_msg_='The open table is '||dsname(tableid)||'.';
```

- In an FSEDIT or FSVIEW SCL program, display on the command line the name of the table that is currently being edited:

```
_msg_='The table being edited is '||dsname()||'.';
```

See Also

"OPEN" on page 576

ENDBLOCK

Closes the window that is created by the BLOCK function

Category: Window

Syntax

CALL ENDBLOCK();

Example

Create a menu that contains four choices. The first row of blocks contains two blocks with the labels Outline and Index. The second row contains two blocks with the labels Compare Files and Calendar. No third row of blocks is displayed. The memory that was used in displaying the menu is freed when the ENDBLOCK routine is executed and the window is closed.

```
INIT:
   choice=block('Writers Toolbox','Main Menu',6,
         'Outline','Index','','',
         'Compare Files','Calendar','','',
         '','','','');
   ...more SCL statements...
return;
MAIN:
   ...more SCL statements...
return;

TERM:
   call endblock();
return;
```

See Also

"BLOCK" on page 263

ENDCATCH

Ends a CATCH statement block

Category: Control Flow

Syntax

ENDCATCH;

Details

The ENDCATCH statement marks the end of a CATCH block.

See Also

"CATCH" on page 268

ENDCLASS

Ends a CLASS statement block

Category: Object Oriented

Syntax

ENDCLASS;

Details The ENDCLASS statement marks the end of a CLASS block in an SCL program. Use ENDCLASS to designate the end of a block of SAS statements that define a class.

See Also

"CLASS" on page 277

ENDLEGEND

Closes the LEGEND window

Category: Legend

Syntax

CALL ENDLEGEND();

Details If the LEGEND window is not currently displayed, the routine has no effect.

For an example of using ENDLEGEND as well as other functions that manipulate the LEGEND window, see "LEGEND" on page 507.

See Also

"POPLEGEND" on page 596

"PUSHLEGEND" on page 603

"PUTLEGEND" on page 606

ENDMETHOD

Ends a METHOD statement block

Category: Modular Programming and Object Oriented

Syntax

ENDMETHOD;

Details The ENDMETHOD statement marks the end of a method block in an SCL program. Use ENDMETHOD with a METHOD statement to indicate a block of statements that can be called by the METHOD routine. When the method block is executed, control returns to the calling program when ENDMETHOD is encountered.

Example

End a METHOD block:

```
METHOD;
   ...SCL statements...
endmethod;
```

See Also

"METHOD" on page 540

ENDPACKAGE

Ends a PACKAGE statement block

Category: Object Oriented

Syntax

ENDPACKAGE;

Details

The ENDPACKAGE statement marks the end of a PACKAGE block. Use ENDPACKAGE to designate the end of a block of ITEM statements that define a package.

See Also

"PACKAGE" on page 585

ENDSUBMIT

Ends statements to be submitted to SAS software for execution

Category: Submit Block

Syntax

ENDSUBMIT;

Details The ENDSUBMIT statement marks the end of a SUBMIT block in an SCL program. Use ENDSUBMIT with SUBMIT to indicate a block of SAS statements to submit to SAS software for execution.

The ENDSUBMIT statement instructs SCL to stop collecting statements in the PREVIEW buffer and to submit the collected statements, based on the options that were specified for the SUBMIT statement.

Example

Use SUBMIT to invoke the PRINT procedure and use ENDSUBMIT to mark the end of the SUBMIT block:

```
submit immediate;
data one;
   do x=1 to 10;
   output;
   end;
run;
proc print;
run;

endsubmit;
```

See Also

"SUBMIT" on page 676

ENDTABLE

Stops the processing of the getrow section of a dynamic extended table

Category: Extended Table

Syntax

CALL ENDTABLE();

Details The ENDTABLE routine stops the processing of the getrow section of a dynamic extended table. A *dynamic extended table* is a table whose maximum number of rows is determined when the program executes. Call the ENDTABLE routine from the getrow section of the SCL program when the end of the extended table has been reached.

Because you can define extended tables only in SAS/AF software, you cannot use ENDTABLE in FSEDIT or FSVIEW programs.

The ENDTABLE routine marks only the end of the table for this invocation of the GETROW label. If the user issues a scroll command, the GETROW label is driven again until ENDTABLE is called. This allows the size of the table to change dynamically.

Example

In this example, data for the extended table comes from the open SAS table that is identified by the value in the variable DSID. The _CURROW_ variable, which identifies the current row of the extended table, specifies which row to fetch. When the value of _CURROW_ exceeds the number of rows in the table, FETCHOBS returns a nonzero value, which indicates that the end of the extended table has been reached. ENDTABLE is then called to stop the processing of the GETROW label.

```
GETROW:
   ...SCL statements...
   if (fetchobs(dsid,_currow_) =-1) then
      call endtable();
   else do;
      ...more SCL statements...
   end;
return;
```

See Also

"CURTOP" on page 324

"SETROW" on page 664

"TOPROW" on page 689

ENDUSECLASS

Ends a USECLASS statement block

Category: Object Oriented

Syntax

ENDUSECLASS;

Details The ENDUSECLASS statement marks the end of a USECLASS block in an SCL program. Use ENDUSECLASS with USECLASS to designate a block of SAS statements that define methods for a class that was previously defined in the Class Editor.

See Also

"USECLASS" on page 698

ENTRY

Receives parameters from the DISPLAY function or routine

Category: Modular Programming and Object Oriented

Syntax

ENTRY <*argument-list*> <RETURN=*data-type*> <OPTIONAL=*argument-list* | <ARGLIST=*arg-list-id* | REST=*rest-list-id*>>;

argument-list
lists one or more sets of arguments, with each set specified as follows:

var-list <: INPUT |UPDATE |OUTPUT> :*data-type*

var-list
lists one or more variables to which the parameter in the corresponding position in the DISPLAY routine or function is passed. For details, see "DISPLAY" on page 350.

INPUT | I
specifies that, at run time, the variable contains the value that is copied from the corresponding parameter of the calling program. However, when the program finishes, the value is not copied back to the calling program. This is equivalent to using CALL NOCHANGE() in the calling program.

UPDATE | U
specifies that, at run time, the variable contains the value that is copied from the corresponding parameter of the calling program. When the program finishes, the value is copied back to that parameter (unless CALL NOCHANGE is specified).

OUTPUT | O
specifies that, when the program finishes, the value is copied back to the corresponding parameter in the calling program. An error condition results if the corresponding parameter in the calling program is a constant, because a constant cannot receive a value.

data-type
specifies the data type of the variable. Any valid SCL data type may be specifid. A named data type (for example, CHAR or LIST) must be preceded by the **:** delimiter. The delimiter is optional for unnamed data types (for example, $).

arg-list-id
contains the identifier for the SCL list that will contain all the arguments passed to the ENTRY statement. This includes all optional arguments.
Type: List

rest-list-id
contains the identifier for the SCL list that will contain all arguments that are passed to the ENTRY statement but are not explicitly specified in *argument-list* for either ENTRY or OPTIONAL=.
Type: List

Details The ENTRY statement receives parameters from the DISPLAY routine or function. It can also return a value if the ENTRY statement contains both the RETURN= option to declare the data type of the returned value and a RETURN statement that specifies either the variable containing the value or the literal value to be returned.

To be compatible with the applications built in earlier releases of SAS software, the **:** delimiter is optional for variables that are assigned unnamed data types (for example,

$), but it is required for variables that are assigned named data types. The following example shows a variety of data type declarations:

```
ENTRY:  char1  :$20
        char2  $20
        char3  :input :char(20)
        char4 char5:char
        num1   :8
        num2   8
        num3   : num
        mylist :list
        myobj  :object
        mybutton :mylib.mycat.button.class return=char;
```

RETURN=*data-type* enables you to return a value to the calling program. An error condition is produced if *data-type* is not the same as the type of data to be returned to the calling program. Use a RETURN statement in the program to specify the value to return.

When there are no options in the ENTRY statement, there is a strict correspondence between DISPLAY parameters and ENTRY statement arguments. The arguments and parameters must agree in number, data type, and relative position. If you pass an incorrect number of parameters or a parameter of the incorrect type, SCL stops executing the program. The correspondence of arguments to parameters is less restrictive when you use the options REST=, ARGLIST=, and OPTIONAL= in the ENTRY statement.

OPTIONAL= enables you to specify a list of optional arguments that are used only if the calling program supplies the corresponding parameters in the DISPLAY parameter list. If the corresponding parameters in the DISPLAY routine are not supplied, then the optional arguments are initialized to missing values.

ARGLIST= and REST= enable you to pass a variable number of parameters to the ENTRY statement. You determine the types and order of the variable arguments. The lists identified by *arg-list-id* and *rest-list-id* are created automatically when the entry is called, and they are deleted automatically when the entry ends. When arrays are passed as parameters, the array is expanded into individual items, and these items are inserted into the *arg-list-id* and *rest-list-id* lists. ARGLIST= and REST= are mutually exclusive, so you can use only one or the other.

The called program can modify all call-by-reference arguments that it receives. However, it cannot modify any call-by-value arguments. For a description of call-by-reference and call-by-value, see “DISPLAY” on page 350 .

By default, values for call-by-reference arguments are returned to the calling program. If you want a called program to receive values but not to return values to its calling program, use the NOCHANGE routine. Or, you can use the INPUT, OUTPUT, or UPDATE option for each variable to specify how its value is passed and returned.

An SCL program with ENTRY statement arguments cannot be executed by itself due to the uninitialized arguments. To test a program that receives parameters via the ENTRY statement, run it with the SCL debugger. The debugger enables you to initialize all the ENTRY arguments before program execution starts.

Examples

Example 1: Returning a Value from the ENTRY Statement B.SCL contains the following ENTRY statement, which uses several numeric arguments:

```
entry x  y z u v :i :num return=num;
INIT:
   total=x+y+z+u+v;
```

```
return(total);
```

A.SCL contains

```
total=display('b.scl',1,2,3,4,5);
put total=;
```

The output of A.SCL is

```
total=15
```

Example 2: Using ENTRY with OPTIONAL= B.SCL contains the following ENTRY statement, which defines multiple character variables:

```
entry x   :char(10)
      y z :num
      optional=u v w :num ;
INIT:
   put x= y= z= u= v= w=;
return;
```

Suppose A.SCL contains

```
call display('b.program','one',2,3,4,5);
```

The output would be

```
X='one' Y=2 Z=3 U=4 V=5 W=.
```

Example 3: Using ENTRY with ARGLIST= B.SCL contains the following ENTRY statement, declaring both numeric, character, and list variables:

```
entry x   :char(10)
      y z :num
      optional=u v   :num
      arglist=mylist;

INIT:
   put x= y = z= u= v=;
   call putlist(mylist);
return;
```

Suppose A.SCL contains

```
call display('b.scl','one',2,3,4,5);
```

The output would be

```
x='one'
y=2
z=3
u=4
v=5('one' 2 3 4 5) [list-id]
```

Example 4: Using ENTRY with ARGLIST= Suppose B.SCL contains

```
entry arglist=mylist;
 INIT:
    call putlist(mylist);
 return;
```

Suppose A.SCL contains

```
call display('b.scl','one',2,3,4,5);
```

The output would be

```
 ('one' 2 3 4 5) [list-id]
```

Example 5: Using ENTRY with REST= B.SCL contains the following ENTRY statement, which declares numeric variables:

```
entry x y :num
      rest=mylist;
INIT:
   put x= y=;
   call putlist(mylist);
 return;
```

Suppose A.SCL contains

```
call display('b.scl',1,2,3,4,5);
```

The output would be

```
x=1
y=2
(3 4 5) [list-id]
```

Example 6: Using ENTRY with OPTIONAL= and REST= B.SCL contains the following ENTRY statement, which declares both numeric and character variables along with parameter Z, which is optional:

```
 entry x y :num optional=z :num rest=mylist;
 INIT:
    put x= y= z=;
    call putlist(mylist);
 return;
```

Suppose A.SCL contains

```
call display('b.scl',1,2,3,4,5);
```

The output would be

```
x=1
y=2
z=3
(4 5) [list-id]
```

See Also

"DISPLAY" on page 350

"NOCHANGE" on page 569

"METHOD" on page 540

"RETURN" on page 615

ENVLIST

Returns the list identifier of an SCL environment list

Category: List

Syntax

list-id=**ENVLIST**(<*envlist-type*>);

list-id
contains the identifier of the SCL environment list.
Type: Numeric or List

envlist-type
specifies which environment list to return:

'G'	returns the identifier for the global environment list.
'L'	returns the identifier for the local environment list. (This is the default.)

Type: Character

Details When the SAS session starts, a global environment list that persists for the entire session is created. When a SAS/AF or FSEDIT application starts executing, a local environment list is created for that application. This local environment list persists for the duration of that application and is available to each program that runs in that application (for example, a program that is invoked by the DISPLAY routine).

Environment lists are special lists that can contain numeric items, character items, and sublist items. You use the item names to fetch the items. You can use the item names to fetch the items. Environment lists provide a means of creating global variables that can be shared among different SCL programs, much like macro variables. However, unlike macro variables, the names in an environment list do not have to be valid SAS names, and the values in an environment list can be other lists, if you want to associate a lot of data with a single name. For example, you can read the contents of a small SAS table into a list and place the contents in the global environment list so that other SCL programs do not have to read the table to fetch data.

You can also insert items that do not have names.

The Local Environment List The local environment list, which is returned by ENVLIST('L'), contains data that is available only to the current application. Each executing application has its own unique local environment list. Both the contents of an application's environment list and the list itself are deleted when the application ends.

The Global Environment List The global environment list, which is returned by ENVLIST('G'), contains data that all SAS applications can share during the same SAS session. The data remains in the global environment list until an SCL program explicitly removes it. Thus, after one application puts data into the global environment list, the application can exit, and another application can fetch the data.

You can insert only global lists into the global environment list. Global lists are created with MAKELIST or MAKENLIST (using the 'G' visibility value), or they can be new lists that have been copied from other global lists. A fatal error results if you try to insert a local list into the global list.

Recommendations for Modifying Environment Lists It is strongly recommended that you insert only named items into environment lists and that you choose names that are unambiguous. If you add items that have simple names, other applications are more likely to unknowingly use the same name and to accidentally overwrite your data.

See Also

"CURLIST" on page 320

"DELLIST" on page 336

"MAKELIST" on page 535

"MAKENLIST" on page 536

ERROR

Reports whether a FRAME entry control or field contains an invalid value

Category: Control or Field

Syntax

rc=**ERROR**(*wvar-name*);

rc
indicates whether the FRAME entry control or field in the window is in error:

1	in error
0	not in error

Type: Numeric

wvar-name
is the name of a FRAME entry control or field in the window. This argument cannot be an expression.
Type: Character

Details Use ERRORON and ERROROFF to set and reset error conditions for the FRAME entry control or field.

The FRAME entry control or field cannot be an element of an array. To report this information for an array element, use FIELD instead.

FRAME entry controls can also use the _inError method.

Examples

Example 1: Using ERROR to Report Invalid Values Specify the CONTROL statement with the ERROR option in the INIT section of the program. The statements in the MAIN section are submitted only if the FRAME entry control OBJ1 is not in error.

```
INIT:
   control error;
return;
MAIN:
   if (error(obj1)=0) and obj1 NE _blank_ then
      submit continue;
         proc print data=&obj1;
         run;
      endsubmit;
   else
```

```
  _msg_='Nothing submitted.  Please correct error.';
  return;
```

Example 2: Generating a Compile Error with ERROR The following sequence generates a compile error because the variable DS is not a window variable but contains the name of a window variable. ERROR expects to be passed the window variable itself.

```
  ds='tablename';
  if (error(ds)) then
     do;
     ...SCL statements to handle the error condition...
     end;
```

See Also

"DISPLAYED" on page 353

"ERROROFF" on page 376

"ERRORON" on page 377

"FIELD" on page 394

"MODIFIED" on page 550

ERROROFF

Clears the error flag on one or more FRAME entry controls or fields

Category: Control or Field

Syntax

ERROROFF *wvar-names* |_ALL_;

wvar-names
specifies one or more window variables for which to turn off the error flag, or **_ALL_** to turn off the error flag for all window variables.
Type: Character

Details An error flag can be set either by attributes that are assigned to fields or FRAME entry controls, or by the ERRORON statement.

Use the following statement to clear the error flag for all FRAME entry controls or all fields in the window:

```
  erroroff _all_;
```

Statements in MAIN do not execute by default if a field is placed in error. Therefore, use a CONTROL statement that specifies the ERROR option to enable ERROROFF to remove the error flag.

The FRAME entry control or field cannot be an element of an array. To remove the error flag for an array element, use FIELD instead.

FRAME entry controls can also use the _erroroff method.

Example

If a user enters an invalid value in the field TABLENAME, this SAS/AF program resets the value of TABLENAME to the default and turns off the error flag. The field TABLENAME is assigned type INPUT, so the procedure checks to see whether the SAS table exists when a user enters a value for TABLENAME.

```
INIT:
   control error;
return;
MAIN:
   if (error(tablename)= 1) then
      do;
         tablename='my.default';
         erroroff tablename;
      end;
return;
```

See Also

"CONTROL" on page 302

"DISPLAYED" on page 353

"ERROR" on page 375

"ERRORON" on page 377

"FIELD" on page 394

ERRORON

Sets the error flag for one or more FRAME entry controls or fields

Category: Control or Field

Syntax

ERRORON *wcol-names* |_ALL_;

wcol-names
specifies one or more window variables for which to turn on the error flag, or **_ALL_** to turn the error flag on for all window variables.
Type: Character

Details To set an error flag for multiple fields, specify the field names following ERRORON, separated by blanks. To set an error flag for all fields in the window, use the following statement:

```
erroron _all_;
```

To clear the error flag for one or more fields, use ERROROFF (see the preceding entry).
In SAS program entries, ERRORON causes the SCL program to execute when a user presses any key the next time the window is displayed. Any fields for which the error

flag is set are marked as modified regardless of whether or not the user has changed the value in the field.

In FSEDIT applications where a field is placed in error with ERRORON, a user can enter a new value and the error status is removed from the field and reset if the error condition is still met. In SAS/AF applications where a field is placed in error with ERRORON, entering a valid value is not enough to remove the error flag. You must use ERROROFF.

The FRAME entry control or field cannot be an element of an array. To set the error flag for an array element, use FIELD instead.

FRAME entry controls can also use the _erroron method.

Example

Suppose your application manipulates a SAS table that contains information about employees and the number of hours they work each week. Because only weekly personnel are paid for overtime, the application should verify that all employees who have reported overtime hours are weekly employees.

```
if (weekly='N' and overtime>0) then
   do;
      erroron overtime;
      _msg_=
     'Only weekly personnel can have overtime.';
      return;
   end;
```

See Also

"CONTROL" on page 302

"DISPLAYED" on page 353

"ERROR" on page 375

"ERROROFF" on page 376

"FIELD" on page 394

EVENT

Reports whether a user has pressed a function key, ENTER key, or mouse button

Category: Keys

Syntax

rc=**EVENT**();

rc

contains the return code for the operation:

1	A function key, ENTER key, or mouse button was pressed since the last event call.

0	A function key, ENTER key, or mouse button was not pressed since the last event call.

Type: Numeric

Details A mouse click registers as an event only when it occurs in a FRAME entry control that responds to mouse events (for example, pushbuttons, check boxes, radio boxes, icons, and blocks).

EVENT is useful when you want your program to continue a task while it waits for user input. For example, your application can read data from an external source and display the results. When a user presses an appropriate key, you can stop processing and handle the request.

Operating Environment Information: OS/390 and CMS
EVENT does not work under OS/390 or CMS, nor on ASCII DRIVER machines. On these systems you should use the attention handler exit that is provided in SCL. Refer to the discussion of the BREAK option for “CONTROL” on page 302. △

Example

Display the date and time until a user presses either ENTER or one of the function keys. The variable DATETIME is a numeric window variable with the format DATETIME17.2. When a user presses ENTER or a function key, the program exits the loop and returns control to the application. In this example, when a user issues a RUN command, the loop resumes.

```
INIT:
   control allcmds;
return;
MAIN:
   if _status_ in ('C','E') then return;
   if (word(1,'U')='RUN') then
      do while(event()=0);
         datetime=datetime();
         refresh;
      end;
return;

TERM:
return;
```

See Also

“CONTROL” on page 302

EVENT

Contains the type of event that occurred on a FRAME entry control

Category: System Variable

Details

EVENT is a character system variable. It is provided automatically by the FRAME entry in SAS/AF, and the SCL compiler automatically creates a space for it in the SCL data vector.

In order to use _EVENT_, you must use the DECLARE or LENGTH statement to declare it as a character variable. If _EVENT_ is not declared, the following error is produced when the _select or _objectLabel method executes:

```
ERROR: Expecting string (P), received SCL number
       (symbol '_EVENT_').
```

EVENT has a valid value only when a window control's _select or _objectLabel method is executing. _EVENT_ can have one of the following values:

`' '`(blank)	Modification or selection
`'D'`	Double click (An 'S' select event always precedes a 'D' double click event.)
`'C'`	Command
`'P'`	Pop-up menu request
`'S'`	Selection or single click

Example

The following _select method prints the value of _EVENT_ when a window control is modified.

```
length _event_ $1;

SELECT: method;
   call super(_self_,_select);
   put _event_=;
endmethod;
```

See Also

"_METHOD_" on page 548

"_SELF_" on page 643

"_STATUS_" on page 671

"_VALUE_" on page 704

EXECCMD

Executes one or more commands when control returns to the application

Category: Command

Syntax

CALL EXECCMD(*cval*);

cval
specifies one or more commands to execute. To specify multiple commands, place a semicolon between each command.
Type: Character

Details The commands are collected until another window is displayed or until SCL has finished executing and control is returned to the procedure. The commands are then submitted to the command-line processor before the next window is displayed or before the current window is redisplayed.

The commands collected with EXECCMD will not be executed if the next window displayed is a host window (such as CATLIST, DIRLIST, LIBLIST, FILELIST or MESSAGEBOX). The commands will be held in a buffer until a non-host selector window is displayed or the current window is redisplayed.

With CONTROL ALWAYS in FSEDIT applications or CONTROL ALLCMDS in other SAS/AF applications, statements in MAIN execute before a command that is issued with CALL EXECCMD. This behavior could introduce an infinite loop. Either execute the EXECCMD routine conditionally or specify the command using EXECCMDI with the NOEXEC parameter.

Example

If the table is empty, add a new row:

```
call execcmd('end');
```

See Also

"EXECCMDI" on page 381

EXECCMDI

Executes one or more global commands immediately before processing the next statement, or executes one non-global command when control returns to the application

Category: Command

Syntax

CALL EXECCMDI(*command*<,*when*>);

command
specifies one or more commands to execute. To specify multiple commands, place a semicolon between each command.
Type: Character

when
specifies when the commands will be executed:

'EXEC'
executes commands in the command buffer immediately. (This is the default.)

'NOEXEC'
executes the specified non-global command when control returns to the application. Global commands are still executed immediately.
Type: Character

Details By default, the EXECCMDI routine immediately executes the specified global command or list of global commands. After executing the command, the program statement that immediately follows the call to EXECCMDI is executed. EXECCMDI is valid only in SCL applications that display a window.

If you specify **EXEC**, which is the default for the *when* argument, then you should issue only windowing environment global commands and full-screen global commands through this routine. Any procedure-specific commands that are executed with EXECCMDI are ignored.

An error is displayed on the message line if the string that is passed to EXECCMDI is not a valid command, but the SCL program is not halted. Any statements that follow the call to the routine are still executed. If multiple commands are specified and one is invalid, none of the remaining commands are executed.

With the NOEXEC option, EXECCMDI allows only one procedure-specific or custom command to be executed. EXECCMDI saves the command in the command buffer and does not execute the command immediately. The program statement that immediately follows the CALL EXECCMDI routine is executed. The command in the command buffer is executed when control returns to the application.

If multiple EXECCMDI routines each have the NOEXEC option specified, then only the command that was issued by the last EXECCMDI routine is executed. The previous commands are cleared.

The NOEXEC option does not alter the way global commands are handled. Global commands are still executed immediately.

With CONTROL ALWAYS in FSEDIT applications or CONTROL ALLCMDS in SAS/AF applications, issuing EXECCMDI with the NOEXEC option from MAIN tells SAS not to execute statements in MAIN again before executing the specified procedure-specific or custom command. This is different from issuing an EXECCMD routine from MAIN, which would execute statements in MAIN again before executing the specified command.

Note: We do not recommend combining EXECCMD and EXECCMDI routines, because the order of execution may be unexpected. △

Examples

Example 1: Using EXECCMDI to Ensure Correct Window Size Ensure that the window is the correct size when the application runs:

```
INIT:
   call execcmdi('zoom off');
return;
```

Example 2: Using EXECCMDI to Confirm a Delete Request From an FSEDIT SCREEN entry, open CONFIRM.FRAME to confirm the delete request before the row is actually deleted:

```
FSEINIT:
   control always;
   length confirm $ 3;
return;

INIT:
return;

MAIN:
   if word(1, 'U') =: 'DEL' then
      do;
 call display('confirm.frame', confirm);
 if confirm =
   'YES' then call execcmdi('delete', 'noexec');
      end;
return;

TERM:
return;
```

CONFIRM.FRAME contains two pushbutton controls, YES and NO, and is controlled by the following program:

```
  entry confirm $ 3;
  YES:
     confirm = 'YES';
     _status_='H';
  return;

  NO:
     confirm = 'NO';
     _status_='H';
   return;
```

See Also

"EXECCMD" on page 380

EXIST

Verifies the existence of a member of a SAS data library

Category: SAS Table

Syntax

rc=**EXIST**(*member-name*<,*member-type*<,*generation*>>);

rc
contains the return code for the operation:

1	The library member exists.
0	Either *member-name* does not exist or *member-type* is invalid.

Type: Numeric

member-name
is the name of the SAS data library member.
Type: Character

member-type
is the type of SAS data library member:

'ACCESS'
indicates an access descriptor that was created using SAS/ACCESS software.

'CATALOG'
indicates a SAS catalog or catalog entry.

'DATA'
indicates a SAS data file. (This is the default.)

'VIEW'
indicates a SAS data view.
Type: Character

generation
is the generation number of the SAS table whose existence you are checking. If *member-type* is not DATA, *generation* is ignored.
Type: Numeric

Details If *member-name* is not specified, EXIST verifies the existence of the member specified by the system variable _LAST_. If *member-type* contains an invalid value, EXIST returns the value 0.

Examples

Example 1: Verifying the Existence of a SAS Table Call the FSEDIT function only if the SAS table specified in the variable TABLENAME exists. If the table does not exist, display a message on the message line.

```
if (exist(tablename)) then call fsedit(tablename);
else _msg_='Table '||tablename||' does not exist.';
```

Example 2: Verifying the Existence of a SAS Data View Verify the existence of the SAS table view TEST.MYVIEW:

```
rc=exist('test.myview','view');
```

Example 3: Determining if a Data Set Generation Exists Determine if the third generation of the data set **work.one** exists:

```
rc=exist('work.one','data',3);
```

See Also

"CEXIST" on page 276

"FEXIST" on page 392

"FILEEXIST" on page 400

FAPPEND

Appends the current record to the end of an external file

Category: External File

Syntax

sysrc=**FAPPEND**(*file-id*<,*cc*>);

sysrc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

file-id
contains the identifier that was assigned when the file was opened. If *file-id* is invalid, the program halts.
Type: Numeric

cc
specifies a carriage-control character:

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 record starts a new line.
Type: Character

Details FAPPEND adds the record currently contained in the File Date Buffer (FDB) to the end of an external file.

Operating Environment Information: OS/390
Records cannot be appended to partitioned data sets. △

Example

Use FAPPEND to append a record to a file:

```
   /* Assign the fileref THEFILE to the physical */
   /* filename that the user entered in the     */
   /* field FNAME and open it in append mode.    */
rc=filename( 'thefile',fname);
fid=fopen('thefile','a');
if (fid>0) then
   do;
         /* Append a new record to the file. */
      rc=fput(fid,'Data for the new record');
      rc=fappend(fid);
      rc=fclose(fid);
   end;
else
   do;
      ...other SCL statements...
   end;
rc=filename('thefile'',''');
```

See Also

"DOPEN" on page 357

"FGET" on page 393

"FOPEN" on page 420

"FPUT" on page 427

"FWRITE" on page 439

"MOPEN" on page 552

FCLOSE

Closes an external file, a directory, or a directory member

Category: External File

Syntax

sysrc=**FCLOSE**(*file-id*);

sysrc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

file-id
contains the identifier that was assigned when the file was opened. A *file-id* value of **-999** closes all files opened with FOPEN. If *file-id* contains an invalid value, the program halts.
Type: Numeric

Example

Close a file after manipulating it:

```
   /* Assign the fileref THEFILE to the physical */
   /* filename that is stored in the variable FNAME      */
   /* and open it in append mode.    */
rc=filename( 'thefile',fname);
```

```
fileid=fopen('thefile');
if (fileid>0) then
   do;
      rc=fread(fileid);
      rc=fclose(fileid);
   end;
else
   do;
      _msg_=sysmsg();
      return;
   end;
rc=filename('thefile','');
```

See Also

"DCLOSE" on page 328

"DOPEN" on page 357

"FOPEN" on page 420

"MOPEN" on page 552

FCOL

Returns the current column position from the File Data Buffer (FDB)

Category: External File

Syntax

col-num=**FCOL**(*file-id*);

col-num
contains the current column position.
Type: Numeric

file-id
contains the identifier that was assigned when the file was opened. If *file-id* contains an invalid value, the program halts.
Type: Numeric

Details Use FCOL in conjunction with FPOS to move the pointer in the FDB and manipulate the data.

Example

Use FCOL and FPOS to set the pointer in the FDB:

```
   /* Assign the fileref THEFILE to the physical  */
   /* filename that the user entered in the field */
   /* FNAME.                                      */
rc=filename( 'thefile',fname);
```

```
fileid=fopen('thefile','o');
if (fileid>0) then do;
       /* Put data into the FDB, get the     */
       /* current column, move the pointer   */
       /* by 1 and add more data to the FDB. */
   record='This is data for the record';
   rc=fread(fileid);
   rc=fput(fileid,record);
   pos=fcol(fileid);
   rc=fpos(fileid,pos+1);
   rc=fput(fileid,'and more data');
   rc=fwrite(fileid);
   rc=fclose(fileid);
end;
rc=filename('thefile','');
```

The record written to the external file is

```
This is data for the record and more data
```

See Also

"FPOS" on page 426

"FPUT" on page 427

"FWRITE" on page 439

FDELETE

Deletes an external file

Category: External File

Syntax

sysrc=**FDELETE**(*fileref*);

sysrc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

fileref
is the fileref that was assigned to the external file to be deleted. The fileref cannot be associated with a list of concatenated filenames or directories. If the fileref is associated with a directory, a PDS or a PDSE, then the directory, PDS, or PDSE must be empty. You must have permission to be able to delete the file or directory.
Type: Character

Details You can use either the FILENAME statement or the FILENAME function in SCL to assign a fileref. Under some operating environments, you can also use system commands to assign filerefs.

Example

Generate a fileref for an external file and assign it to the variable FREF. Then call FDELETE to delete the file and call the FILENAME function again to deassign the fileref.

```
length fref $ 8;
fref =_blank_;
   /* Assign a fileref generated by the system */
   /* to the physical filename that is stored  */
   /* in the variable FNAME.                   */
 rc=filename(fref,fname);
if (rc=0) and (fexist(fref)) then
   rc=fdelete(fref);
rc=filename(fref,'');
```

See Also

"FEXIST" on page 392

"FILENAME" on page 403

FETCH

Reads the next nondeleted row from a SAS table into the Table Data Vector (TDV)

Category: SAS Table

Syntax

sysrc=**FETCH**(*table-id*<,'NOSET'>);

sysrc
contains the return code for the operation:

0	successful
>0	not successful
<0	the operation was completed, but a warning or a note was generated. If the row is locked, it is still fetched (read in) but a *sysrc* of **_SWNOUPD** is returned.
–1	the end of the table was reached

Type: Numeric

table-id
is the identifier that was assigned when the table was opened. If *table-id* is invalid, the program halts.
Type: Numeric

'NOSET'
prevents the automatic passing of SAS table column values to SCL variables even if the SET routine has been called.
Type: Numeric

Details FETCH skips rows that have been marked for deletion. When a WHERE clause is active, the function reads the next row that meets the WHERE condition.

If the SET routine has previously been called, the values for any table columns that are also window variables or SCL variables for the application are automatically passed from the TDV to the SCL Data Vector (SDV). To temporarily override this behavior so that fetched values are not automatically copied to the SDV, use the NOSET option.

Example

Fetch the next row from the SAS table MYDATA. If the end of the table is reached or if an error occurs, SYSMSG retrieves the appropriate message and displays it on the message line.

```
INIT:
   tableid=open('mydata','i');
return;

MAIN:
   rc=fetch(tableid);
   if rc then _msg_=sysmsg();
   else
      do;
         ...more SCL statements...
      end;
return;

TERM:
   rc=close(tableid);
return;
```

See Also

"APPEND" on page 251

"FETCHOBS" on page 390

"GETVARC and GETVARN" on page 450

"LOCATEC and LOCATEN" on page 527

"PUTVARC and PUTVARN" on page 609

"SET" on page 646

"UPDATE" on page 697

FETCHOBS

Reads a specified row from a SAS table into the Table Data Vector (TDV)

Category: SAS Table

Syntax

sysrc=**FETCHOBS**(*table-id*,*row-number*<,*options*>);

sysrc
contains the return code for the operation:

0	successful
>0	not successful
<0	the operation was completed, but a warning or a note was generated. If the row is locked, it is still fetched (read in) but a *sysrc* of **`_SWNOUPD`** is returned.
–1	the end of the table was reached

Type: Numeric

table-id
is the identifier that was assigned when the table was opened. If *table-id* is invalid, the program halts.
Type: Numeric

row-number
is the number of the row to read.
Type: Numeric

options
is one or both of the following options, separated by blanks:

`'ABS'`
specifies that the value of *row-number* is absolute; that is, deleted rows are counted.

`'NOSET'`
prevents the automatic passing of SAS table column values to SCL variables even if the SET routine has been called.
Type: Character

Details If SET has previously been called, the values for any table columns that are also window variables or SCL variables for the application are automatically passed from the TDV to the SCL Data Vector (SDV) with FETCHOBS. You can use NOSET in the FETCHOBS function to temporarily override this behavior so that fetched values are not automatically copied to the SDV. FETCHOBS treats the row value as a relative row number unless the ABS option is specified.

The row value may or may not coincide with the physical row number on disk. For example, the function skips rows that have been marked for deletion. When a WHERE clause is active, the function counts only rows that meet the WHERE condition. If *row-number* is less than 0, the function returns an error condition. If *row-number* is greater than the number of rows in the SAS table, an 'End of file' warning is returned.

Example

Fetch the tenth row from the SAS table MYDATA. If the end of the table is reached, a message to that effect is displayed on the message line. If an error occurs, the SYSMSG function retrieves the error message and displays it on the message line.

```
rc=fetchobs(mydata,10);
if (rc=-1) then
  _msg_='End of table has been reached.';
if (rc ne 0) then _msg_=sysmsg();
```

See Also

"APPEND" on page 251

"FETCH" on page 389

"GETVARC and GETVARN" on page 450

"LOCATEC and LOCATEN" on page 527

"PUTVARC and PUTVARN" on page 609

"SET" on page 646

"UPDATE" on page 697

FEXIST

Verifies the existence of the external file that is associated with the specified fileref

Category: External File

Syntax

rc=**FEXIST**(*fileref*);

rc
contains the return code for the operation:

1	successful
0	not successful, or there was no logical assignment for the fileref

Type: Numeric

fileref
is the fileref that was assigned to the external file.
Type: Character

Details You can use either the FILENAME statement or the FILENAME function in SCL to assign filerefs. Under some operating systems, you can also use system commands to assign filerefs. Use FILEEXIST to verify the existence of a file based on its physical name.

Example

Verify the existence of an external file for a fileref that the user enters in the field for the window variable FREF. A message informs the user whether the file exists.

```
 if (fexist(fref)) then
_msg_=
   'The file does exist.';
else
   _msg_=sysmsg();
```

See Also

"EXIST" on page 383

"FILEEXIST" on page 400

"FILENAME" on page 403

"FILEREF" on page 406

"PATHNAME" on page 586

FGET

Copies data from the File Data Buffer (FDB)

Category: External File

Syntax

sysrc=**FGET**(*file-id*,*cval*<,*length*>);

rc

contains the return code for the operation:

0	successful
–1	the end of the FDB was reached, or no more tokens were available.

Type: Numeric

file-id

is the identifier that was assigned when the file was opened. If *file-id* is invalid, the program halts.

Type: Numeric

cval

is a character variable to hold the data.

Note: This parameter is an update parameter. See "Input, Output, and Update Parameters" on page 34 for more information. △

Type: Character

length

specifies how many characters to retrieve from the FDB.

Type: Numeric

Details FGET copies data from the FDB into a character variable. If *length* is specified, then only the specified number of characters is retrieved (or the number of characters remaining in the buffer, if that number is less). If *length* is omitted, then 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 "FSEP" on page 435 for more information about delimiters.)

After FGET is executed, the column pointer is automatically moved to the next "read" position in the FDB.

Example

Read the first record in the file specified by the user and copy the first token into the variable THESTRING.

```
      /* Assign the fileref THEFILE to the physical */
      /* filename that is stored in the    */
      /* variable FNAME and open it in append mode.  */
   rc=filename( 'thefile',fname);
   fileid=fopen('thefile');
   if (fileid>0) then
      do;
            /* Read the first record, retrieve the  */
            /* first token of the record and store  */
            /* it in the variable THESTRING.        */
         rc=fread(fileid);
         rc=fget(fileid,thestring);
         put thestring;
         rc=fclose(fileid);
      end;
   rc=filename('thefile','');
```

See Also

"FPOS" on page 426

"FREAD" on page 430

"FSEP" on page 435

FIELD

Performs an action on or reports the state of FRAME entry widgets or fields

Category: Widget or Field

Syntax

rc=**FIELD**(*action*<,*wvar-name-1*<,*wvar-name-2*
<,*wvar-name-3*>>>);

rc
contains the return code for the operation. The return value is dependent on the action.
Type: Numeric

action
is an action from the list of actions described in "Values for the *action* Argument" on page 395.
Type: Character

wvar-name-1, wvar-name-2, wvar-name-3
are character columns or expressions whose values are the names of one or more FRAME entry widgets or fields in a window, separated by spaces. At least one name is required for all actions except ALARM, BLOCKCUROFF, BLOCKCURON, CURSCREEN, HOME, NSCREEN, and SMOOTHSCRL.
Type: Character

Values for the *action* Argument

The following list contains the values that you can specify for *action*. The list also includes the corresponding methods that can be used in FRAME entries, which are documented with FRAME entry classes in the online Help for SAS/AF software.

'ALARM'
sounds a bell. This action has a corresponding SCL statement.

'BLOCKCUROFF'
turns the block cursor off so that fields or text entry widgets are not highlighted when a user tabs or moves the cursor to them.

'BLOCKCURON'
turns the block cursor on, which causes input fields to be highlighted when the cursor is on the text entry widget or field.

'COLOR *color <attribute>*'
changes the color and display attribute of a field, text entry widget, or text label widget in the window. Colors are: BLACK, BLUE, BROWN, CYAN, GRAY, GREEN, MAGENTA, ORANGE, PINK, RED, WHITE, and YELLOW. Attributes are: NONE, BLINKING, HIGHLIGHT, HIREV, REVERSE, and UNDERLINE. SASCOLOR window elements can be used for *color*. If you specify a SASCOLOR window element for *color*, then *attribute* is not allowed, because the SASCOLOR window element already contains a display attribute.

'COLUMN'
returns the column where a FRAME entry widget or field is located. This option is valid only in SAS/AF software. FRAME entry widgets can also use the _column method.

'CUROBS'
returns the current row number for FSEDIT and FSVIEW for the specified field.

'CURSOR'
positions the cursor in the FRAME entry widget or field. If more than one field or widget is specified, the cursor is positioned in the last one specified. This action has a corresponding SCL statement. FRAME entry widgets can also use the _cursor method.

'CURSCREEN'
returns the current screen number. For SAS/AF software, this is valid only for multipage PROGRAM entries. For FSEDIT, it reports which screen of a multiscreen application is displayed.

'DISPLAYED'
returns the total number of FRAME entry widgets or fields that are visible, or 0 if none of them are currently displayed. For example, if you pass three field names and two are visible, then *rc* is 2. This action has a corresponding SCL function. FRAME entry widgets can also use the _isDisplayed and _isHidden methods.

'ERROR'
returns the total number of FRAME entry widgets or fields that are in error, or 0 if none of the specified fields are in error. For example, if you pass two field names and one is in error, then *rc* is 1. This action has a corresponding SCL function. FRAME entry widgets can also use the _inError method.

'ERROROFF'
removes the error status from one or more FRAME entry widgets or fields. This action has a corresponding SCL statement. FRAME entry widgets can also use the _erroroff method.

'ERRORON'
turns on the error status for one or more FRAME entry widgets or fields. Turning on the error status prevents users from ending the SAS/AF application or from leaving the current row in FSEDIT. The error status also highlights the field or widget, using the error color and display attributes that were assigned in the Attribute (or ATTR) window. This action has a corresponding SCL statement. FRAME entry widgets can also use the _erroron method.

'GETOBS'
reports whether a formula is being executed in an FSVIEW application because a column is being read. If *rc* is 1, then the formula is being executed because a column is being read. If *rc* is 0, then the formula is being executed because a column has been modified. If you are on a new row, *rc* is always 1. This is the opposite of PUTOBS.

'HOME'
moves the cursor to the command line. This entry has a corresponding SCL statement.

'ICON *icon-number*'
assigns a number for an icon that represents the field if it is a pushbutton in PROGRAM entries. This option is valid only in SAS/AF software. FRAME entry widgets can use the _setIcon method.

'MODIFIED'
returns the total number of FRAME entry widgets or fields that were modified, or 0 if none of them were modified. For example, if you pass two field names and both were modified, then *rc* is 2. This action has a corresponding SCL function. FRAME entry widgets can also use the _isModified method.

'NSCREEN'
returns the number of screens (for FSEDIT applications) or the number of panes (for SAS/AF applications).

'PROTECT'
protects one or more FRAME entry widgets or fields. This prevents a user from modifying the FRAME entry widget or field. This action has a corresponding SCL statement. FRAME entry widgets can also use the _protect method.

'PROTECTED'
reports whether a FRAME entry widget or field is protected. FRAME entry widgets can also use the _isProtected method.

'PUTOBS'
reports whether a formula is being executed in an FSVIEW application because a column has been modified. If *rc* is 1, then the formula is being executed because a column has been modified. If *rc* is 0, then the formula is being executed to read a column. If you are on a new row, *rc* is always 0. This is the opposite of GETOBS.

'ROW'
returns the row where a FRAME entry widget or field is positioned. This option is valid only in SAS/AF software. FRAME entry widgets can also use the _row method.

'SMOOTHSCRL'
sets smooth scrolling to **ON**, **OFF**, or **TOGGLE**. Allows smooth scrolling when users drag the thumb in the scroll bar. When smooth scrolling is on, the getrow sections of AF extended tables are called while the thumb is dragged. In the FSVIEW procedure, the display is refreshed while the thumb is dragged. When smooth

scrolling is turned off, the redisplay is deferred until the thumb is released. By default, smooth scrolling is off for SAS/AF and on for FSVIEW.

If you do not specify ON or OFF, SMOOTHSCRL is toggled from OFF to ON or from ON to OFF.

'UNPROTECT'
unprotects one or more fields or FRAME entry widgets. This enables a user to modify the field or FRAME entry widget. This action has a corresponding SCL statement. FRAME entry widgets can also use the _unprotect method.

Details The FIELD function combines the functionality of the field statements (CURSOR, DISPLAYED, ERROROFF, ERRORON, PROTECT, and so on). It also provides additional widget over the fields.

At least one window column name is required for all actions except ALARM, BLOCKCUROFF, BLOCKCURON, CURSCREEN, HOME, NSCREEN, and SMOOTHSCRL.

The smooth scrolling action enables you to turn on, turn off, or toggle the scrolling mode.

Examples

Example 1: Using FIELD to Allow Smooth Scrolling Allow smooth scrolling:

```
rc=field ('smoothscrl', 'on');
```

Example 2: Using FIELD to Check for Error Status of Fields Create the array FLDNAMES and pass its elements to the FIELD function to check the error status of the fields. If necessary, move the cursor to the field that contains invalid data.

```
array fldnames{*} $ 8 ('tablename','colname','list',
                       'x','y');
do i=1 to dim(fldnames);
   if (field('error',fldnames{i})) then
      do;
         _msg_='Field name '||
         fldnames{i}||' is bad.';
         rc=field('cursor',fldnames{i});
         return;
      end;
end;
```

Example 3: Using FIELD to Turn on an Error Flag Turn on the error flag for FIELD1 and FIELD2:

```
rc=field('erroron','field1','field2');
```

Example 4: Using FIELD to Change a Field's Color Change the color of FIELD1:

```
rc=field('color blue','FIELD1');
```

Example 5: Using FIELD to Change Color and Display Attributes for a Field Change FIELD1's color and display attributes:

```
rc=field('color red reverse','field1');
```

Example 6: Using FIELD to Specify a SASCOLOR Window Element Specify FIELD1's color, using the name of a SASCOLOR window element:

```
rc=field('color foreground','field1');
```

See Also

"ALARM" on page 250

"CURSOR" on page 323

"DISPLAYED" on page 353

"ERROR" on page 375

"ERROROFF" on page 376

"ERRORON" on page 377

"HOME" on page 459

"MODIFIED" on page 550

"PROTECT" on page 602

"UNPROTECT" on page 695

FILEDIALOG

Displays a selection window that lists external files

Category: External File

Syntax

rc=**FILEDIALOG**(*dialog-type,filename<,default-file <,default-dir<,filter-1<. . . ,filter-11<,description-1 . . .<description-11>>>>>>*);

rc
contains the return code for the operation:

–1	A user cancelled without selecting a file.
0	Either *dialog-type* is **OPEN** and the file exists, or *dialog-type* is **SAVEAS** and the file does not exist.
1	*Dialog-type* is **SAVEAS**, the file exists, and the user wants to replace the file.
2	*Dialog-type* is **SAVEAS**, the file exists, and a user wants to append to the file.

Type: Numeric

dialog-type
specifies the type of dialog window to open. An invalid type specification produces an error and halts the program. Types are

'AGGREGATE'
lists aggregate storage areas — for example, directories, partitioned data sets, or MACLIBs.

'LIBRARY'
lists SAS data libraries.

'OPEN'
lists files that a user can open.

'SAVEAS'
lists files that a user can write to.
Type: Character

filename
is the fully qualified name of the selected file, including the directory.

Note: This parameter is an update parameter. See "Input, Output, and Update Parameters" on page 34 for more information. △
Type: Character

default-file
is the name of the file (without directory information) to display as selected when the dialog window opens. If you specify a null string (''), the file that was selected last is the default file.
Type: Character

default-dir
is the directory whose files are listed when the dialog window opens. If you specify a null string (''), the directory that was selected last is the default directory.
Type: Character

filter1 … filter11
are up to 11 name specifications to narrow the list of files to display — for example, *.html. The number of filter arguments is host specific. If you do not supply a filter, the list contains all files in *default-dir*.
Type: Character

description1 … description11
are up to 11 descriptions, one for each *filter*, to make the dialog window more informative for application users. If no descriptions are supplied, a default description is displayed for each specified filter. If you provide a description for any filter, then you must supply a description for each filter that you specify.
Type: Character

Details Depending on the values of *default-dir* and *filter*, *default-file* may not be in the list of files displayed. Therefore, *default-file* will not be selected.

An error condition is produced if you supply a description for at least one filter but fail to supply a description for each specified filter.

Operating Environment Information: The formats of the files and filter parameters are all host specific. The UNIX and MicroSoft Windows platforms use all of the passed filters. The Macintosh platform ignores the filter argument. All other platforms use only the first filter that is passed. △

Examples

- Enable a user to select a file to open, and see whether the user cancelled the window:

```
rc=filedialog('saveas',selfile,'autoexec.sas',
              '/sas','*.sas');
   /* Process the selected file  */
select(rc);
   when(0) put 'New file selected';
   when(1) put 'REPLACE an existing file';
```

```
      when(2) put 'APPEND to an existing file';
      when(-1) put 'User pressed cancel';
      otherwise put 'ERROR occurred';
   end;
```

- □ Display a list of filenames that have .SAS, .HTML, and .GIF extensions, and provide descriptions for these filters:

```
rc=filedialog('open',selfile,'','',
  '*.sas','*.html','*.gif','','','','','','','','',
  'SAS Files','Web Pages','Images');
```

See Also

"FILELIST" on page 401

FILEEXIST

Verifies the existence of an external file, a directory, or a SAS data library by its physical name

Category: External File

Syntax

sysrc=**FILEEXIST**(*filename*);

rc
contains the return code for the operation:

1	The external file exists.
0	The external file does not exist.

Type: Numeric

filename
is the name that identifies the external file to the host operating system. The *filename* specification varies according to the operating system.
Type: Character

Details FILEEXIST verifies the existence of an external file, a directory, or a SAS data library.

Although your system utilities may recognize partial physical filenames, you must always use fully qualified physical filenames with FILEEXIST.

Example

Verify the existence of an external file whose filename the user enters in the field for the window variable FNAME. Display a message on the message line to tell the user whether the file exists.

```
if (fileexist(fname)) then
   _msg_='The external file '||fname||' exists.';
else
   _msg_=sysmsg();
```

See Also

"EXIST" on page 383

"FEXIST" on page 392

"FILENAME" on page 403

"FILEREF" on page 406

"PATHNAME" on page 586

FILELIST

Displays a host selection window that lists the currently assigned filerefs, and returns user selections

Category: Selection List

Syntax

selections=**FILELIST**(<*sel-excl*<,*message*<,*autoclose* <,*num-sel*>>>>);

selections

contains the user's selections, or a blank if no fileref was selected. Multiple selections are separated by blanks. By default, *selections* is 200 bytes long. To accomodate values longer than 200 bytes, explicitly declare *selections* with a longer length.

Type: Character

sel-excl

specifies which filerefs to include in the selection window. Specify as

- □ one or more filerefs that have been assigned for the current SAS session. Use spaces to separate multiple filerefs.
- □ an asterisk (`'*'`) or a null string (`''`) to display all the filerefs that are defined for the current SAS session.
- □ a NOT sign (¬ or ^) followed by one or more filerefs, to display all filerefs except those listed after the NOT sign. For example, '^ MYFILE1 MYFILE2' displays all defined filerefs except MYFILE1 and MYFILE2.

Type: Character

message

is the text for a message to display above the selection list. The default message tells users to make up to the number of selections specified in *num-sel*.

Type: Character

autoclose

is an obsolete argument but is retained for compatibility with earlier releases. If you want to specify a value for *num-sel*, then specify `''`as a placeholder for this argument.

Type: Character

num-sel

is the maximum number of items a user can select from the list. To display the list for information purposes only (no selections allowed), specify 0. To specify an unlimited number of selections, use a value such as 9999 that is larger than the

number of available selections. A user cannot make a number of selections that exceeds the number of items in the list.
Type: Numeric

Details The selection list displays both filerefs and the corresponding physical names of the external files to which the filerefs are assigned, but only the selected fileref is returned.

If you omit all the arguments for FILELIST (for example, **`selections=filelist();`**), the selection list window contains all filerefs that have been assigned in the current SAS session.

You can provide default values that will be initially selected when the fileref selection list is displayed. To do this, assign the values to the *selections* variable before calling FILELIST.

If a user closes the selection list window without making a selection, FILELIST returns a blank value unless there was an initial value for the *selections* variable before FILELIST was called.

Selections from the window can be returned in the current result list, if one is available. The current result list is a special SCL list that is automatically filled with the values that are selected from a selection list. To use a current result list, use the MAKELIST function to create the list, and use the CURLIST function to designate it as the current result list. The current result list must exist before you call the FILELIST function.

When FILELIST is invoked, the current result list is cleared. After FILELIST is invoked, the current result list contains the following named items:

TAG
identifies the list as one that was created by FILELIST.
Type: Character

COUNT
contains the number of selected filerefs, or 0 if a user makes no selections or issues a CANCEL command in the list window.
Type: Numeric

FILEREF
contains the name of each selected fileref. There is a FILEREF element for each selected fileref.
Type: Character

FILENAME
contains the physical name of the external file for each selected fileref. There is a FILENAME element for each selection made.
Type: Character

Because some engines support mixed-case filenames, FILELIST now retains the cases of the returned selected items. This may cause your application to fail if your application contains code that assumes the returned selection is uppercased. For example,

```
if (filelist(dsid, 'TESTNDX')='NDXVAR')
```

must be changed to

```
if (upcase(filelist(dsid, 'TESTNDX'))='NDXVAR'
```

If the application cannot be modified, you may need to specify the VALIDVARNAME=V6 system option when you run the application to ensure that the selections returned from the FILELIST function will be uppercased.

Examples

Example 1: Displaying Specified Filerefs Open a window that displays a list of all defined filerefs except for LISTNUM.

```
select=filelist('^listnum');
```

Example 2: Using a Current Result List for Multiple User Selections Open a window that displays a list of all defined filerefs except LISTNUM. Users can make up to five selections. The selections are retrieved from the current result list.

```
listid=makelist();
rc=curlist(listid);
select=filelist('^listnum',' ',' ',5);
n=getnitemn(listid,'COUNT');
do i=1 to n;
   fileref=getnitemc(listid,'FILEREF',i);
   physname=getnitemc(listid,'FILENAME',i);
   put fileref= physname=;
end;
```

See Also

"CATLIST" on page 269

"DIRLIST" on page 347

"LIBLIST" on page 513

FILENAME

Assigns or deassigns a fileref for an external file, a directory, an output device, or a catalog entry

Category: External File

Syntax

sysrc=**FILENAME**(*fileref*,*filename*<,*device* <,*host-options*<,*dir-ref*>>>);

sysrc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

fileref
is the fileref to assign. A blank *fileref* (`''`) causes an error condition. If the fileref is an SCL character variable that has a blank value, a fileref will be generated for you.
Type: Character

filename
is the physical name of an external file. Specifying a blank *filename* deassigns one that was previously assigned.
Type: Character

device
is the type of device if the fileref points to something other than a physical file:

CATALOG
a catalog entry

DUMMY
output to the file is discarded

FTP
the file transfer protocol (FTP) access method

GTERM
graphics on the user's terminal

NAMEPIPE
a named pipe

PIPE
an unnamed pipe

Note: Some host operating systems do not support pipes. △

PLOTTER
an unbuffered graphics output device

PRINTER
a printer or printer spool file

SOCKET
the Transmission Control Protocol/Internet Protocol (TCP/IP) socket access method

TERMINAL
the user's terminal

TAPE
a tape drive

URL
the URL access method
Type: Character

host-options
are host-specific details such as file attributes and processing attributes. *Host-options* can also be used to specify device options. For example, they could include output destinations, and CATALOG, FTP, URL, TCPIP, and SOCKET options. For details about host and device options for the FILENAME statement, see *SAS Language Reference: Dictionary* and the SAS documentation for your operating environment.
Type: Character

dir-ref
is the fileref assigned to the directory or partitioned data set in which the external file resides.
Type: Character

Details The name associated with the file or device is called a *fileref* (file reference name). Other SCL functions that manipulate external files and directories require that the files be identified by a fileref rather than by a physical filename. A system-generated fileref is not displayed in the FILENAME window.

The association between a fileref and a physical file lasts only for the duration of the current SAS session or until you use FILENAME to change or discontinue the association. You can de-assign filerefs by specifying a null string for the *filename* argument in FILENAME.

For more information about the arguments that you can use with FILENAME, see *SAS Language Reference: Dictionary* and the SAS documentation for your operating environment.

Operating Environment Information: The term *directory* in this description refers to an aggregate grouping of files that are managed by the host operating system. Different host operating systems identify such groupings with different names, such as directory, subdirectory, MACLIB, or partitioned data set. See the SAS documentation for your operating environment for details.

Under some operating systems, you can also use system commands to assign filerefs. Depending on the operating system, FILENAME may be unable to change or de-assign filerefs that are assigned outside of a SAS session.

See the SAS documentation for your host operating system for information about the system-dependent options that you can specify for *options*. △

Examples

Example 1: Assigning a Fileref Assign the fileref MYFILE to an external file:

```
   /* Assign fileref MYFILE to the physical */
   /* filename stored in the variable FNAME   */
rc=filename('myfile',fname);
if (rc ne 0) then
   _msg_=sysmsg();
```

Example 2: Using a System-Generated Fileref Assign a system-generated fileref, stored in the variable FNAME, to the file whose physical name is in the control FNAME:

```
fname=' ';
   /* Assign a system-generated fileref to the */
   /* filename stored in the variable FNAME      */
rc=filename(fname,fname);
if (rc) then
   _msg_=sysmsg();
else
   do;
      ...more SCL statements...
   end;
    /* De-assign the fileref */
rc=filename('myfile','');
```

Example 3: Making an External File Accessible to a Client Assign a fileref to an external file:

```
rc=filename('sharedfl','\ABC\XYZ\AUTOEXEC.SAS);
```

Example 4: Assigning a Fileref for a Pipe File Assign a fileref for a pipe file with the output from the UNIX command LS, which lists the files in the directory /u/myid:

```
rc=filename('myfile','ls /u/myid','pipe');
```

See Also

"FEXIST" on page 392

"FILEEXIST" on page 400

"FILEREF" on page 406

"PATHNAME" on page 586

FILEREF

Verifies that a fileref has been assigned for the current SAS session or process

Category: External File

Syntax

sysrc=**FILEREF**(*fileref*);

sysrc
contains the return code for the operation:

0	Both the fileref and the external file exist.
< 0	The fileref has been assigned, but the file that it points to does not exist.
> 0	The fileref has not been assigned.

Type: Numeric

fileref
is the fileref to be validated (up to eight characters).
Type: Character

Details A negative return code indicates that the fileref exists but that the physical file associated with the fileref does not exist. A positive, nonzero value indicates that the fileref has not been assigned.

A fileref can be assigned to an external file by using the FILENAME statement or the FILENAME function in SCL. Under some operating systems, you can also use system commands to assign filerefs. See the SAS documentation for your operating environment.

Examples

Example 1: Determining Whether a Fileref Has Been Assigned to an External File Test whether the fileref MYFILE is currently assigned to an external file. A system error message is issued if the fileref is not currently assigned.

```
if (fileref('myfile') > 0) then _msg_=sysmsg();
```

Example 2: Determining Whether a Fileref and Its File Exist Test the fileref MYFILE to determine whether the fileref is assigned and whether the file that it refers to exists:

```
if (fileref('myfile') ne 0) then _msg_=sysmsg();
```

See Also

"FEXIST" on page 392

"FILEEXIST" on page 400

"FILENAME" on page 403

"PATHNAME" on page 586

FILLIST

Fills an SCL list with text and data

Category: List

Syntax

sysrc=**FILLIST**(*type*,*source*,*list-id*<,*attr-list-id*<,*description*>>);

sysrc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

type
specifies the type of file or data source named in *source* and one or more options to use:

'CATALOG<(*options*)>**'**
specifies that *source* names a catalog entry.

'FILE<(*options*)>**'**
specifies that *source* names an external file.

'FILEREF<(*options*)>**'**
specifies that *source* names a fileref that has been assigned to an external file.

'SASICONS<(*numbers*)>**'**
specifies the numbers for icons that are provided with SAS software. (When you specify SASICONS, *source* is ignored. Specify a null argument `''`for *source*.)

'SEARCH'
specifies catalog names in the current search path (*source* is ignored). Use the SEARCH function to define the search path, or specify `''`for *source*.

The available options are described in "*Type* Options" on page 408. Separate multiple options with blanks. For example, to fill a list from an external print file and to strip carriage-control characters, specify **FILE(PRINT STRIPCC)** for *type*.
Type: Character

source
is a catalog entry (specified as *libref.catalog.entry.type*), an external file, or a fileref.
Type: Character

list-id
contains the identifier of the list to fill. An invalid *list-id* produces an error condition.
Type: Numeric or List

attr-list-id
contains the identifier of the list to fill with text attribute *source* information when *type* is **CATALOG** and the entry type is LOG, OUTPUT, or SOURCE. An invalid *attr-list-id* produces an error condition.
Type: Numeric

description
is the name of the variable in which you want to store the text of the catalog entry description. This argument is ignored if *type* is **FILE** or **FILEREF**.

Note: This parameter is an update parameter. See "Input, Output, and Update Parameters" on page 34 for more information. △
Type: Character

Type Options

ADDCC
adds default host carriage-control characters. Used with *type* **FILE**, **FILEREF**, and **CATALOG** and with catalog entry types **LOG**, **OUTPUT**, and **SOURCE**.

PRINT
designates an external file as a PRINT file (uses the host carriage-control characters). Used with *type* **FILE** and **FILEREF**.

STRIPCC
removes carriage-control characters. Used with *type* **FILE**, **FILEREF**, and **CATALOG** and with catalog entry types **LOG**, **OUTPUT**, and **SOURCE**.

TRIM
trims trailing blanks. Used with *type* **FILE**, **FILEREF** and **CATALOG** and with catalog entry types **LOG**, **OUTPUT**, and **SOURCE**. **TRIM** is used if you want to use FILLIST to fill a list with items that contain trailing blanks and then remove the blanks so that they will not be displayed in a pop-up menu that is produced by POPMENU.

Details Each line of text in the source file is placed in a separate character item of the list identified by *list-id*. The number of items in the filled list is determined by the number of lines of text. All SCL lists must be created with MAKELIST before you call FILLIST. FILLIST automatically clears the lists before it fills the lists.

Data from the external file or catalog entry cannot exceed the maximum length of a character value in an SCL list item, which is 32,766 characters.

External Files If *type* is **FILE**, then *source* is the name of an external file. If *type* is **FILEREF**, then *source* is a SAS fileref. FILLIST reads each record of the file into a separate character item.

SLIST Catalog Entries If *type* is **CATALOG** and the catalog entry type in *source* is **SLIST**, then the types of the items in the filled list are determined by the saved list, and they may be character strings, numbers, or other lists. All item attributes and names are duplicated, as are the list attributes. However, the list identifier numbers are different.

LIST Catalog Entries If *type* is **CATALOG** and the catalog entry type in *source* is **LIST**, FILLIST reads the contents of a SAS/AF LIST entry into *list-id*. The list contains either all numeric or all character items, depending on the contents of the LIST entry. The attribute list contains the following named values, which are all character type:

INFORMAT
: is the SAS informat that was specified in the LIST entry.

FORMAT
: is the SAS format that was specified in the LIST entry.

MESSAGE
: is the error message that was specified in the LIST entry.

CAPS
: reports whether the LIST entry has the CAPS attribute. **Y** or **N**.

SORTED
: reports whether the LIST entry has the SORT attribute. **Y** or **N**.

NOHONORCASE
: reports whether the LIST entry has the CASE-INSENSITIVE attribute. **Y** or **N**.

TYPE
: is the TYPE attribute that was specified in the LIST entry. **N** for numeric, **C** for character.

JUST
: is the JUST attribute that was specified in the LIST entry. **L**, **R**, **C**, or **N** for left, right, center, or none.

SOURCE, OUTPUT, and LOG Catalog Entries If *type* is **CATALOG** and the entry type is **OUTPUT, LOG**, or **SOURCE**, the first character in each list item contains a FORTRAN carriage-control character: 1 means that a new page starts with this line. See **STRIPCC** above. **ADDCC** converts all carriage-control characters to **' '**(blank).

If *type* is **CATALOG** and the entry type is **OUTPUT, LOG**, or **SOURCE**, then any text attributes (such as color and display attributes), are read one element per line into *attr-list-id*, if it is specified. These attributes consist of a character item for each line of text. Each character item contains one character for each character in the line, plus a prefix descriptor character. The prefix character is **T** for a title line, **H** for a header line, or **D** for a data line. The other characters represent the text display attributes and color, as described in the tables below.

Do not confuse text attributes (color, display, and so on) with list attributes that are specified with SETLATTR.

The attribute list that is filled by FILLIST contains one item for each line of text from the SAS catalog entry. The attribute string for each line has one character for each character of text. Each attribute character represents the SAS windowing environment color and display attribute. Not all display devices support all colors.

Color attributes are represented as follows:

Color	**Value**	**Color**	**Value**
BLUE	’10’x	WHITE	’70’x
RED	’20’x	ORANGE	’80’x
PINK	’30’x	BLACK	’90’x
GREEN	’40’x	MAGENTA	’A0’x

Color	Value	Color	Value
CYAN	'50'x	GRAY	'B0'x
YELLOW	'60'x	BROWN	'C0'x

Display attributes are represented as follows:

Attribute	Value
NONE	'00'x
HIGHLIGHT	'01'x
UNDERLINE	'02'x
BLINK	'04'x
REVERSE	'08'x

You combine the color and display attributes by adding them together. For example, you can specify GREEN UNDERLINE by adding '40'x to '02'x to yield '42'x. To assign GREEN UNDERLINE to the first 4 characters of a string, you could use a statement like:

```
str = '42424242'x;
```

See also “STRATTR” on page 675, which creates attribute strings.
You can use GETITEMC or POPC to retrieve an item from this list.
An error condition is produced if

- either list has the NOUPDATE, NUMONLY, or FIXEDLENGTH attribute
- any item in either list cannot be removed because it has the NODELETE attribute.

Examples

Example 1: Reading Text and Attributes Into a List Suppose you have an OUTPUT entry named FINANCE.REPORTS.MONTHLY.OUTPUT that contains the text "Net:($45,034)" on line 45. The text **`Net:`** is white with no highlight attributes, whereas the text **`($45,034)`** is red reverse. The following statements read the text and attributes and print line 45.

```
INIT:
   text_list=makelist();
   attr_list=makelist();
   rc=fillist('CATALOG',
      'FINANCE.REPORTS.MONTHLY.OUTPUT',
      text_list,attr_list);
   text=substr(getitemc(text_list,45),2);
   attr=substr(getitemc(attr_list,45),2);
   len=compress(put(2*length(text), 4.));
   attrhex=putc(attr,'$HEX'||len||'.');
   put attr;
   put text;
   put attrhex;
return;
```

Note: SUBSTR removes the carriage-control characters. △

This example produces the following output:

```
ppppp(((((((((
Net: ($45,034)
7070707070282828282828282828
```

The line of text consists of five white characters with no attributes, represented by the attribute value '70'x, followed by nine red reverse characters, represented by '28'x.

Example 2: Performing a Recursive List Copy The following statements perform an operation similar to a recursive list copy:

```
rc=savelist('CATALOG','WORK.TEMP.MYLIST.SLIST',
   mylist);
new_list=makelist();
rc=fillist('CATALOG','WORK.TEMP.MYLIST.SLIST',
   new_list);
rc=delete('WORK.TEMP.TEMP.SLIST','CATALOG');
```

Lists that are saved in a permanent catalog with SAVELIST can persist across SAS sessions.

Example 3: Reading and Printing Out Data and Attributes from LIST Entries Consider two LIST entries: SASUSER.DATA.A.LIST, which contains some character data, and SASUSER.DATA.DATES.LIST, which contains formatted numeric data. The following program reads the data and attributes from these entries and uses PUTLIST to print the results.

```
INIT:
items=makelist();
attrs=makelist();
rc=fillist('catalog','sasuser.data.a.list',
   items,attrs);
call putlist(items,'A.LIST contents:',0);
call putlist(attrs,'A.LIST attributes:',0);
rc=fillist('catalog','sasuser.data.dates.list',
   items,attrs);
call putlist(items,'DATES.LIST contents:',0);
call putlist(attrs,'DATES.LIST attributes:',0);
rc=dellist(items);
rc=dellist(attrs);
return;
```

The output for these entries may look like this:

```
A.LIST contents:('THIS     '
                 'IS       '
                 'A        '
                 'LIST     '
                 'ENTRY    '
                 'WITH     '
                 'EIGHT    '
                 'ITEMS    '
                 )[5]
A.LIST attributes:(INFORMAT=''
                   FORMAT=''
                   MESSAGE=''
                   CAPS='Y'
```

```
                      SORTED='N'
                      NOHONORCASE='Y'
                      TYPE='C'
                      JUST='L'
                      )[7]
DATES.LIST contents:(1765
                      11162
                      11813
                      12072
                      )[5]
DATES.LIST attributes:(INFORMAT='DATE.'
                        FORMAT='DATE.'
                        MESSAGE=''
                        CAPS='Y'
                        SORTED='Y'
                        NOHONORCASE='N'
                        TYPE='N'
                        JUST='L'
                        )[7]
```

Note: [5] and [7] are the list identifiers that were assigned when this example was run and may be different each time the example is run. △

See Also

"SAVELIST" on page 629

FINFO

Returns a file information item

Category: External File

Syntax

item-value=**FINFO**(*file-id*,*info-item*);

item-value
contains the value of the file parameter, or a blank if *info-item* is invalid.
Type: Character

file-id
is the identifier that was assigned when the file was opened. If *file-id* is invalid, the program halts.
Type: Numeric

info-item
specifies which file information item to retrieve.
Type: Character

Details FINFO returns the value of a system-dependent information item for an external file. The information that is available for files depends on the operating system.

FOPTNUM determines how many system-dependent information items are available. FOPTNAME determines the names of the available items.

Example

Assign the fileref MYFILE to an external file. Then open the file and determine whether LRECL is one of the available information items. If the value of the variable CHARVAL is nonblank, then a value for LRECL (logical record length, an attribute used on some host systems), is displayed to the user.

```
rc=filename('myfile',fname);
fid=fopen('myfile');
charval=finfo(fid,'lrecl');
if (charval=' ') then
_msg_= 'The LRECL attribute is not available.';
else
   _msg_='The LRECL for the file is||charval||'.';
rc=fclose(fid);
rc=filename('myfile','');
```

See Also

"DINFO" on page 346

"FOPEN" on page 420

"FOPTNAME" on page 421

"FOPTNUM" on page 423

FKEYNAME

Returns the name of the specified function key

Category: Keys

Syntax

key-name=**FKEYNAME**(*key-number*);

key-name
contains a function key name as listed in the KEYS window. Function key names vary according to the device.
Type: Character

key-number
is the number that corresponds to the order in which the keys are displayed in the KEYS window.
Type: Numeric

Details The *key-number* argument identifies a key by its ordinal position in the KEYS window, not by its label. For example, if the first key in the KEYS window is named PF1, use a 1 rather than PF1 for the *key-number* argument to identify that key. To retrieve the corresponding key definitions, use GETFKEY.

You can use this function only in entries that have a DISPLAY window.

Example

Return the name of function key 12:

```
keyname=fkeyname(12);
```

See Also

"GETFKEY" on page 440

"NUMFKEYS" on page 574

"SETFKEY" on page 649

FLDATTR

Changes the color and display attributes of a field, text entry widget, or text label widget to those stored in an attribute string

Category: Widget or Field

Syntax

rc=**FLDATTR**(*wcol-name*,*string*);

rc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

wcol-name
specifies the field, text entry widget, or text label widget to be changed.
Type: Character

string
specifies the color and display attributes to apply and the starting and ending character positions within the field.
Type: Character

Details You can use STRATTR or FILLIST to generate the attribute string. You can also generate the attribute string by assigning hexadecimal values directly to the string.
Color attributes are represented as

Color	Value	Color	Value
BLUE	'10'x	WHITE	'70'x
RED	'20'x	ORANGE	'80'x
PINK	'30'x	BLACK	'90'x

Color	Value	Color	Value
GREEN	’40’x	MAGENTA	’A0’x
CYAN	’50’x	GRAY	’B0’x
YELLOW	’60’x	BROWN	’C0’x

Display attributes are represented as

Attribute	Value
NONE	’00’x
HIGHLIGHT	’01’x
UNDERLINE	’02’x
BLINK	’04’x
REVERSE	’08’x

To preserve a color, use the special hexadecimal value ’F0’x. To preserve a display attribute, use ’0F’x. To preserve both the color and display attribute, add the two special characters together (’FF’x).

For programs with extended tables you must call this function in the getrow section of your SCL program.

FRAME entry widgets can also use the _setColorStr method.

Example

Change the first half of the field, ABC, to red reverse.

```
str=strattr('red','reverse',1,mlength(abc)/2);
rc=fldattr('abc',str);
```

Suppose the FRAME text entry widget, OBJ1, is BLUE REVERSE. To change the third through the seventh character positions of OBJ1 to yellow, you must initialize the first two characters of the attribute string to ’FF’x, then assign YELLOW in the third through seventh characters. You can assign YELLOW to the attribute string either by using STRATTR or by assigning the hexadecimal values directly to the string.

```
st ='FFFF6060606060'x;
rc=fldattr('obj1',str);
```

The previous example could have been written as follows:

```
str='FFFF'x;
str=strattr('yellow','',3,5);
rc=fldattr('obj1',str);
```

You can also use the REPEAT function to initialize a string.

```
str=repeat('FF'x,2 );
str=strattr('yellow','',3,5);
rc=fldattr('obj1',str);
```

See Also

"FLDCOLOR" on page 416

"STRATTR" on page 675

FLDCOLOR

Changes the color and display attributes of a field, text entry widget, or text label widget to those stored in an attribute string

Category: Widget or Field

Syntax

rc=**FLDCOLOR**(*wcol-name*,*color*,*attribute*,*start*,*length*);

rc

contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

wcol-name

specifies the field, text entry widget, or text label widget to be changed.

Type: Character

color

specifies either a color name, or `''`to retain the current color. Colors are BLACK, BLUE, BROWN, CYAN, GRAY, GREEN, MAGENTA, ORANGE, PINK, RED, WHITE, and YELLOW. SASCOLOR window elements can also be used for *color*.

Type: Character

attribute

specifies either a display attribute, or `''`to retain the current attribute. Attributes are NONE, BLINKING, HIGHLIGHT, HIREV, REVERSE, and UNDERLINE. If you specify a SASCOLOR window element for *color*, then *attribute* is ignored because the SASCOLOR window element contains a display attribute. However, you must specify a placeholder (`''`) for *attribute* when you specify arguments after it.

Type: Character

start

specifies the position in the field at which to begin applying the specified color and display attributes.

Type: Numeric

length

specifies the number of positions to which the specified color and display attributes are to be applied.

Type: Numeric

Details FRAME entry widgets can also use the _setColor method.

To change the color for the entire field or FRAME entry widget, you can use the FIELD function.

Examples

Example 1: Changing the Color and Attributes of Character Positions Change the color of the third through seventh character positions in field ABC to red, and change the display attribute of those positions to high intensity:

```
rc=fldcolor('abc','red','highlight',3,5);
```

Example 2: Changing the Color of a Field Change the color of a field, but leave the attributes the same:

```
rc=fldcolor('abc','red','',3,7);
```

Example 3: Using a SASCOLOR Window Element to Change the Color of a Field Change the color of a field, using a SASCOLOR window element:

```
rc=fldcolor('abc','foreground','',3,7);
```

See Also

"FIELD" on page 394

"FLDATTR" on page 414

"STRATTR" on page 675

FNOTE

Identifies the last record that was read

Category: External File

Syntax

note-id=**FNOTE**(*file-id*);

note-id
contains the identifier assigned to the last record that was read. The *note-id* value is used by the FPOINT function to reposition the file pointer on a particular record. SCL programs should not modify the value of the *note-id* variable.
Type: Numeric

file-id
is the identifier that was assigned when the file was opened. If *file-id* is invalid, the program halts.
Type: Numeric

Details You can use FNOTE like a bookmark, marking the position in the file so that your application can later use FPOINT to return to that position.

FNOTE is limited to noting 1,000 records. When that limit is reached, the program halts. To free the memory that is associated with each note identifier, use DROPNOTE.

Example

Assign the fileref THEFILE to an external file. Then attempt to open the file. If the file is successfully opened, indicated by a positive value in the variable FID, then read the records, use the variable NOTE3 to save the position of the third record that is read, and then later use FPOINT to point to NOTE3 to update the file. After the record is updated, close the file.

```
     /* Assign the fileref THEFILE to the        */
     /* filename stored in the variable FNAME    */
     /* and open it in UPDATE mode.              */
  rc=filename('thefile',fname);
  fileid=fopen('thefile','u');
  if (fileid>0) then do;
        /* Read the first record. */
     rc=fread(fileid);
        /* Read the second record. */
     rc=fread(fileid);
        /* Read the third record. */
     rc=fread(fileid);
        /* Note the position of third record. */
     note3=fnote(fileid);
        /* Read the fourth record. */
     rc=fread(fileid);
        /* Read the fifth record. */
     rc=fread(fileid);
        /* Point to the third record. */
     rc=fpoint(fileid,note3);
        /* Read the third record. */
     rc=fread(fileid);
        /* Copy the new text to the FDB. */
     rc=fput(fileid,'New text');
        /* Update the third record with data in the FDB. */
     rc=fwrite(fileid);
        /* Close the file. */
     rc=fclose(fileid);
  end;
     /* Deassign the fileref.  */
  rc=filename('thefile','');
```

See Also

"DROPNOTE" on page 361

"FPOINT" on page 425

"FREAD" on page 430

"FREWIND" on page 431

FONTSEL

Opens the selector window for host fonts or for portable fonts

Category: Selection List

Syntax

newfontlist-id=**FONTSEL**(*oldfontlist-id*,*font-selector*);

newfontlist-id
contains the identifier of the list that contains the selected font family, size, weight, and style.
Type: Numeric

oldfontlist-id
contains the identifier of the list that contains the font information for the selection list. An invalid *oldfontlist-id* produces an error condition. This list can be empty.
Type: Numeric

font-selector
specifies which font selector window to open:

'Y'
the host font selector window

'N'
the portable font selector window

'H'
the portable font selector window, displaying only the hardware fonts

' '(blank)
the default font selector window
Type: Character

Details If *oldfontlist-id* is not empty, then the selector window opens with the font family, size, weight, and style selections that are specified in the list. If *oldfontlist-id* is an empty list, then the selector window opens with the default selections for font family, size, weight, and style. The *newfontlist-id* identifier contains information about the font family, size, weight, and style that the user selected.

The host font selector window enables a user to select fonts that are available on the host in an environment-specific way. The portable font selector window enables a user to select a portable font specification, which is used to find the closest match among fonts that are available on a host. The host font selector window can also be opened from the portable font selector window by using the System button.

For more information about how to use the font information that is returned, see the documentation for the extended text entry class in SAS/AF software and its _setFont method.

To change the default font selector window, use the SAS system option MULTENVAPPL, which is described in *SAS Language Reference: Dictionary*.

Example

Make a FRAME entry with a pushbutton control named PUSHBTN and a Version 8 text entry control named TEXTENTRY1. Clicking on the pushbutton executes the code to display the portable font selector window. Change the font selector value from **N** to **Y** to use the host font selector window.

```
INIT:
    fontid=makelist();
return;
PUSHBTN:
    fontid=fontsel(fontid,'n');
    rc=putlist(fontid,'FONT',1);
    textentry1.font=fontid;
return;
```

```
TERM:
    rc=dellist(fontid);
return;
```

See Also

"MAKELIST" on page 535

FOPEN

Opens an external file

Category: External File

Syntax

file-id=**FOPEN**(*fileref*<,*open-mode*<,*record-length*
<,*record-format*>>>);

file-id
contains the identifier for the file, or 0 if the file could not be opened.
Type: Numeric

fileref
is the fileref that is assigned to the external file.
Type: Character

open-mode
specifies the type of access to the file:

'A'	APPEND mode, which allows writing new records after the current end of the file.
'I'	INPUT mode, which allows reading only. (This is the default.)
'O'	OUTPUT mode, which defaults to the OPEN mode specified in the host option in the FILENAME statement or function. If no host option is specified, it allows writing new records at the beginning of the file. If the file exists, its contents are overwritten, destroying any previous contents. If the file does not exist, it is created.
'S'	Sequential input mode, which is used for pipes and other sequential devices such as hardware ports.
'U'	UPDATE mode, which allows both reading and writing records.
'W'	Sequential output mode, which is used for pipes and other sequential devices such as hardware ports.

Type: Character

record-length
is the logical record length of the file. To use the existing record length for the file, either specify a length of 0 or do not provide a value here.
Type: Numeric

record-format

is the record format of the file. To use the existing record format, do not specify a value here.

'B'	The data is to be interpreted as binary data.
'D'	Use the default record format.
'E'	Use an editable record format.
'F'	The file contains fixed-length records.
'P'	The file contains printer carriage-control characters in a host-dependent record format.
'V'	The file contains variable-length records.

Type: Character

Details

CAUTION:

Use OUTPUT mode with care. Opening an existing file in OUTPUT mode 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 can then be used to identify the open file to other functions. You must associate a fileref with the external file before calling the FOPEN function.

In SCL you can assign filerefs by using either the FILENAME statement or the FILENAME function. Under some operating systems, you can also use operating system commands to assign filerefs.

Operating Environment Information: OS/390

For OS/390 data sets that have the VBA record format, specify 'P' for the *record-format* argument. △

Examples

- Assign the fileref MYFILE to an external file. Then attempt to open the file for input, using all defaults:

```
rc=filename('myfile',fname);
fid=fopen('myfile');
```

- Attempt to open the file for input, this time not using defaults:

```
fid=fopen('file2','o',132,'e');
```

See Also

"DOPEN" on page 357

"FCLOSE" on page 386

"FILENAME" on page 403

"FILEREF" on page 406

"MOPEN" on page 552

FOPTNAME

Returns the name of an item of information for a file

Category: External File

Syntax

item-name=**FOPTNAME**(*file-id*,*item-num*);

item-name
contains the name of the information item, or a blank if an error occurred.
Type: Character

file-id
contains the identifier that was assigned when the file was opened. If *file-id* is invalid, the program halts.
Type: Numeric

item-num
is the number of the information item.
Type: Numeric

Details The number, value, and type of available information items depend on the operating system.

Example

Retrieve the system-dependent file information items and write them to the log:

```
length name $ 8;
rc=filename('myfile',fname);
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;
end;
rc=fclose(fid);
rc=filename('myfile','');
```

The example produces the following output:

```
File attribute LRECL has a value of 256.
```

See Also

"DINFO" on page 346

"DOPTNAME" on page 358

"DOPTNUM" on page 359

"FINFO" on page 412

"FOPEN" on page 420

"FOPTNUM" on page 423

"MOPEN" on page 552

FOPTNUM

Returns the number of information items that are available for an external file

Category: External File

Syntax

num-items=**FOPTNUM**(*file-id*);

num-items
contains the number of information items that are available.
Type: Numeric

file-id
contains the identifier that was assigned when the file was opened. If *file-id* is invalid, the program halts.
Type: Numeric

Details The number, value, and type of available information items depend on the operating system.

Use FOPTNAME to determine the names of the items that are available for a particular operating system. Use FINFO to retrieve the value of a particular information item.

Example

Open the external file that has the fileref MYFILE and determine how many system-dependent file information items are available:

```
fileid=fopen('myfile');
infonum=foptnum(myfile);
```

See Also

"DINFO" on page 346

"DOPTNAME" on page 358

"DOPTNUM" on page 359

"FINFO" on page 412

"FOPTNAME" on page 421

FORMAT

Verifies that the specified format is valid

Category: Formatting

Syntax

rc=**FORMAT**(*format*,*type*);

rc
contains the return code for the operation:

1	successful
0	not successful

Type: Numeric

format
contains a format that is either supplied by SAS or created using the FORMAT procedure. The format name must contain a period (for example, dollar6. or dollar8.2).
Type: Character

type
specifies the type of the format:

`'C'`	character
`'N'`	numeric.

Type: Character

Details If the specified format is not known to the SAS session, then the operation is unsuccessful. The function verifies that valid widths are specified for formats.
See *SAS Language Reference: Dictionary* for details about formats.

Examples

Example 1: Verifying the Validity of a Character Format Assume that you want to use the $CHAR12. format and to verify that $CHAR12. is a valid character format. (If the format name is valid, then the value returned to the variable RC is 1.)

```
rc=format('$char12.','c');
```

Example 2: Verifying the Validity of a Numeric Format Verify that **5.6** is not a valid format for numeric values. (If it is not valid, then the value returned to the variable RC is 0.)

```
rc=format('5.6','n');
```

See Also

"INFORMAT" on page 479

FPOINT

Positions the "read" pointer on the next record to be read

Category: External File

Syntax

sysrc=**FPOINT**(*file-id*,*note-id*);

sysrc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

file-id
contains the identifier that was assigned when the file was opened. If *file-id* is invalid, the program halts.
Type: Numeric

note-id
contains the identifier that was assigned by the FNOTE function.
Type: Numeric

Details Use FNOTE to provide the *note-id* value that identifies the record. 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.

Example

Assign the fileref MYFILE to an external file. Then attempt to open the file. If the file is opened successfully, then read the records and use NOTE3 to store the position of the third record read. Later, point back to NOTE3 to update the file, closing the file afterward.

```
     /* Assign the fileref MYFILE to the physical */
     /* filename stored in the variable FNAME     */
     /* and open it in UPDATE mode.               */
rc=filename('myfile',fname);
fileid=fopen('myfile','u');
```

```
if (fileid>0) then do;
      /* Read the first record. */
   rc=fread(fileid);
      /* Read the second record. */
   rc=fread(fileid);
      /* Read the third record. */
   rc=fread(fileid);
      /* Note the position of the third record. */
   note3=fnote(fileid);
      /* Read the fourth record. */
   rc=fread(fileid);
      /* Read the fifth record. */
   rc=fread(fileid);
      /* Point to the third record. */
   rc=fpoint(fileid,note3);
      /* Read the third record. */
   rc=fread(fileid);
      /* Copy the new text to the FDB. */
   rc=fput(fileid,'new text');
      /* Write data in the FDB to the third record. */
   rc=fwrite(fileid);
      /* Close the file. */
   rc=fclose(fileid);
end;
   /* Clear the fileref. */
rc=filename('myfile','');
```

See Also

"DROPNOTE" on page 361

"FNOTE" on page 417

"FREAD" on page 430

"FREWIND" on page 431

FPOS

Sets the position of the column pointer in the File Data Buffer

Category: External File

Syntax

sysrc=**FPOS**(*file-id*,*nval*);

sysrc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

file-id
contains the identifier that was assigned when the file was opened. If *file-id* is invalid, the program halts.
Type: Numeric

nval
is the number of the column at which to set the pointer.
Type: Numeric

Details If the specified position is past the end of the current record, then the size of the record is increased appropriately. However, in a fixed block file or a 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.

Example

Assign the fileref THEFILE to an external file and then attempt to open the file. If the file is opened successfully, as indicated by a positive value in the variable FID, then place data into the file's buffer at column 12, write the record, and close the file.

```
   /* Assign the fileref THEFILE to the physical */
   /* filename stored in the variable FNAME      */
   /* and open it in append mode.                */
rc=filename('thefile',fname);
fileid=fopen('thefile','o');
if (fileid>0) then do;
    rc=fread(fileid);
    dataline='This is some data.';
       /* Position at column 12 in the FDB. */
    rc=fpos(fileid,12);
       /* Put the data in the FDB. */
    rc=fput(fileid,dataline);
       /* Write the record. */
    rc=fwrite(fileid);
       /* Close the file. */
    rc=fclose(fileid);
end;
   /* Clear the fileref. */
rc=filename('thefile','');
```

See Also

"FCOL" on page 387

"FPUT" on page 427

FPUT

Moves data to the File Data Buffer (FDB) for an external file, starting at the FDB's current column position

Category: External File

Syntax

sysrc=**FPUT**(*file-id*,*cval*<, *length*>);

rc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

file-id
contains the identifier that was assigned when the file was opened. If *file-id* is invalid, the program halts.
Type: Numeric

cval
is the data to be moved to the FDB.
Type: Character

length
is the length of the string to move to the FDB. If *length* is greater than *cval*, then the string is padded with blanks when it is moved. If *length* is less than *cval*, the string is truncated when it is moved. If *length* is less than 1, the program halts.
Type: Numeric

Details The unformatted value of *cval* is passed to FPUT. The number of bytes moved to the FDB is determined by the length of the variable, or by the value of *length*, if *length* is specified. The value of the column pointer is then incremented to one position past the end of the new text.

Example

Move data to the FDB and write it to the external file:

```
   /* Assign the fileref THEFILE to the physical */
   /* filename stored in the variable FNAME      */
   /* and open it in append mode.                */
rc=filename('thefile',fname);
fileid=fopen('thefile','a');
if (fileid>0) then
   do;
      thestring='This is some data.';
      rc=fput(fileid,thestring);
      rc=fwrite(fileid);
      rc=fclose(fileid);
   end;
else
   _msg_=sysmsg();
rc=filename('thefile','');
```

See Also

"FNOTE" on page 417

"FPOINT" on page 425

"FPOS" on page 426

"FWRITE" on page 439

FRAME

Contains the identifier of either the FRAME entry that contains the control or the FRAME entry that is being used as a method

Category: System Variable

Details _FRAME_ is a system variable that is provided automatically by the FRAME entry in SAS/AF software. A space is automatically created for it in the SCL data vector (SDV). _FRAME_ contains the identifier of the FRAME entry that contains a component. Otherwise, it contains the identifier of the FRAME that is running.

You can use this variable to send methods to a FRAME entry from a control's method. For example, a control method can send a _refresh method to the FRAME entry, causing the FRAME entry to refresh its display.

Example

Suppose the entry FIRST.FRAME contains an icon. The icon's _select method is defined to run the SAS/AF FRAME entry SECOND.FRAME, which contains the following program:

```
INIT:
      /* Send a method to the current FRAME */
   _CFRAME_._setMsg('Running the Select method');
return;
TERM:
      /* Send a method to the FRAME that */
      /* contains the icon              */
   _FRAME_._setMsg('Select method has finished.');
return;
```

When FIRST.FRAME displays and a user selects the icon, SECOND.FRAME displays with the message "Running the Select method". After the user ends from SECOND.FRAME, FIRST.FRAME displays the message "Select has finished." This is accomplished by sending the _setMsg method to _CFRAME_ (the FRAME entry that is currently running) in the INIT section and by sending _setMsg to _FRAME_ (the FRAME entry that contains the icon) in the TERM section.

See Also

"_CFRAME_" on page 277

FREAD

Reads a record from an external file into the File Data Buffer (FDB)

Category: External File

Syntax

sysrc=**FREAD**(*file-id*);

sysrc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

file-id
contains the identifier that was assigned when the file was opened. If *file-id* is invalid, the program halts.
Type: Numeric

Details The position of the file pointer is automatically updated after the read operation so that successive FREAD functions read successive file records.
Use FNOTE, FPOINT, and FREWIND to position the file pointer explicitly.

Example

Assign the fileref MYFILE to an external file and attempt to open the file. Read each record from the file and list it in the LOG window.

```
   /* Assign the fileref MYFILE to the physical */
   /* filename stored in the variable FNAME     */
   /* and open it.                              */
rc=filename('myfile',fname);
fileid=fopen('myfile');
if (fileid>0) then
   do while(fread(fileid)=0);
      rc=fget(fileid,c,200);
      put c;
   end;
rc=fclose(fileid);
rc=filename('myfile','');
```

See Also

"FGET" on page 393

"FNOTE" on page 417

"FPOINT" on page 425

"FREWIND" on page 431

FREWIND

Positions the file pointer at the beginning of the file

Category: External File

Syntax

sysrc=**FREWIND**(*file-id*);

sysrc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

file-id
contains the identifier that was assigned when the file was opened. If *file-id* is invalid, the program halts.
Type: Numeric

Details FREWIND has no effect on a file that was opened with sequential access.

Example

Assign the fileref THEFILE to an external file. Then open the file and read 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 is stored in the File Data Buffer (FDB). The first token is retrieved and is stored in the variable VAL.

```
   /* Assign the fileref THEFILE to the physical */
   /* filename stored in the variable FNAME      */
   /* and open it.                               */
rc=filename('thefile',fname);
fileid=fopen('thefile');
do while (rc ne -1);
      /* Read a record. */
   rc=fread(fileid);
end;

   /* Reposition the pointer at the beginning of */
   /* the file.                                  */
if rc= -1  then rc=frewind(fileid);
```

```
      /* Read the first record. */
   rc=fread(fileid);
      /* Read the first token into VAL. */
   rc=fget(fileid,val);
   put val= ;
   rc=fclose(fileid);
   rc=filename('thefile','');
```

See Also

"FGET" on page 393

FRLEN

Returns the size of the last record read, or, if the file is opened for output, returns the current record size

Category: External File

Syntax

length=**FRLEN**(*file-id*);

length
contains the length of the current record if the file is opened for output. Otherwise, it is the length of the last record read.
Type: Numeric

file-id
is the identifier that was assigned when the file was opened. If *file-id* is invalid, the program halts.
Type: Numeric

Example

Open the file identified by the fileref THEFILE. Determine the minimum and maximum lengths of records in the external file, and write the results to the LOG window.

```
      /* Assign the fileref THEFILE to the physical */
      /* filename stored in the variable FNAME      */
      /* and open it.                               */
   rc=filename('thefile',fname);
   fileid=fopen('thefile');
   min=0;
   max=0;
   if (fread(fileid)=0) then do;
       min=frlen(fileid);
       max=min;
       do while(fread(fileid)=0);
          reclen=frlen(fileid);
          if (reclen>max) then max=reclen;
```

```
        if (reclen<min) then min=reclen;
      end;
      rc=fclose(fileid);
   end;
   put min= max=;
```

See Also

"FCLOSE" on page 386

"FOPEN" on page 420

"FREAD" on page 430

FSEDIT

Displays a SAS table by row

Category: SAS Table

Syntax

CALL FSEDIT(*table-name*<,*screen-name*
<,*open-mode*<,*row-number*>>>);

table-name
is the SAS table to display, in the form <*libref.*>*member-name*<(*data-set-options*)>. If you omit the libref, the default WORK library is used.

You can add a list of SAS data set options in parentheses following the table name. All data set options are valid except FIRSTOBS= and OBS=. See *SAS Language Reference: Dictionary* for a list of SAS data set options and their descriptions.

Type: Character

screen-name
is a SCREEN entry for the FSEDIT session. (Screen entries are SAS catalog entries of type SCREEN that define custom features for the FSEDIT session.) For *screen-name,* use the form <*libref.*>*catalog-name*<.*entry-name*<.SCREEN>>. A one- or two-level name is interpreted as a catalog name, and the default screen entry name, FSEDIT.SCREEN, is assumed. (A one-level name is interpreted as a catalog in the default SAS data library, WORK.) If the specified catalog does not already exist, it is created.

If the screen entry does not already exist, a new screen entry is not created unless the user issues a MODIFY command during the FSEDIT session.

If you want to use predefined SAS table labels, use an equal sign (=) for *screen-name.* (A modified SCREEN entry is not saved.) Column names are used for any fields that lack labels.

Type: Character

open-mode
specifies the type of access to the SAS table:

'ADD'
adds a new blank row to the table, then opens the FSEDIT window with the new row displayed for editing.

'BROWSE'
opens the FSBROWSE window for reading rows.

'EDIT'
opens the FSEDIT window for editing rows. (This is the default.)

'NEW'
opens the FSEDIT NEW window for creating the specified SAS table as a new table, then opens the FSEDIT window for entering values into the new table.
Type: Character

row-number
is the first row to be displayed when the FSEDIT or FSBROWSE window is opened. This argument is ignored unless the value of *open-mode* is **'EDIT'** or **'BROWSE'**.
Type: Numeric

Details The FSEDIT routine calls the FSEDIT procedure, which opens the FSEDIT window to display the specified SAS table. You can specify BROWSE for *open-mode* to open the FSBROWSE window for browsing the table instead. You can optionally specify the name of a screen entry to provide a custom display and the number of the row to be displayed when the window is opened.

If you want to specify *open-mode* or *row-number* but do not want to specify a screen entry, use a null string ('') for *screen-name*.

For more information about the commands that are available in the FSEDIT procedure, see *SAS/FSP Software Procedures Guide*.

Examples

- Open a SAS table named PERSONAL (in the default SAS data library WORK) for editing:

```
call fsedit('personal');
```

- Open the SAS table PERSONAL in the library MASTER for editing, using a custom SCREEN entry named PER1 in the catalog MASTER.DISPLAY:

```
call fsedit ('master.personal','master.display.per1');
```

- Open a SAS table named MASTER.PERSONAL for browsing, using the default FSBROWSE window display:

```
call fsedit('master.personal',' ','browse');
```

- To display the predefined labels that are associated with the SAS table columns instead of the column names, specify an equal sign (=) for the *screen-name* argument, as in the following example:

```
call fsedit('master.personal','=','browse');
```

Specify SAS data set options by enclosing them within parentheses immediately following the SAS table name in the *table-name* argument, as in the following examples.

- Open a SAS table named MASTER.PERSONAL and subset the rows based on the value that is entered for the numeric column SCRNUM:

```
call fsedit('master.personal(where=(num='||
             put(scrnum,5.)||'))');
```

- Open a SAS table named MASTER.PERSONAL and subset the rows based on the value that is entered for the character column SCRNAME:

```
call fsedit('master.personal(where=(name='||
             quote(scrname)||'))');
```

See Also

"NEW" on page 561

FSEP

Sets the token delimiters for the FGET function

Category: External File

Syntax

sysrc=**FSEP**(*file-id*,*delimiter*);

sysrc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

file-id
specifies the identifier that was assigned when the file was opened. If *file-id* is invalid, the program halts.
Type: Numeric

delimiter
specifies the token delimiter that separates items in the File Data Buffer (FDB). If multiple characters are specified, each character is considered a delimiter. The default delimiter is a blank.
Type: Character

Example

Suppose the external file contains data in this form:

```
John J. Doe,Male,25,Weight Lifter
```

Note that each field is separated by a comma.

Read the file identified by the fileref THEFILE, using the comma as a separator, and write the values for NAME, GENDER, AGE, and WORK to the LOG window:

```
fileid=fopen('thefile');
rc=fsep(fileid,',');
sysrc=fread(fileid);
rc=fget(fileid,cval);
do while (rc ne -1);
   put cval=;
   rc=fget (fileid,cval);
end;
rc=fclose(fileid);
```

The output of the program is

```
cval=John J. Doe
cval=Male
cval=25
cval=Weight Lifter
```

See Also

"FGET" on page 393

"FREAD" on page 430

FSLIST

Displays an external file for browsing

Category: External File

Syntax

CALL FSLIST(*file*<,*options*>);

file
is the fileref or physical filename of the external file. A physical name must be enclosed in quotation marks.
Type: Character

options
specifies one or more carriage-control options for formatting the display, with multiple options separated by blanks and enclosed in one set of quotation marks:

'CC'
Use the host operating system's native carriage-control characters.

'FORTCC'
Use FORTRAN-style carriage-control characters.

'NOCC'
Treat carriage-control characters as regular text. (This is the default.)
For CC or FORTCC, you can also specify an overprinting control option:

'OVP'
Print the current line over the previous line when the overprint code is encountered. The OVP option is valid only if the CC or FORTCC option is also specified. The default is NOOVP.

'NOOVP'
Ignore the overprint code and print each line from the file on a separate line of the display.

If you use the FORTCC option, the first column of each line in the external file is not displayed. The character in this column is interpreted as a carriage-control code. Under some operating systems, FORTRAN-style carriage control is the native carriage control. For these systems, the FORTCC and CC options produce the same behavior.

Under some operating systems, the CC option is the default for print files.
Type: Character

Details The FSLIST routine calls the FSLIST procedure, which opens the FSLIST window to display an external file for interactive browsing. This routine provides a convenient method for examining the information stored in an external file.

For more information about the commands that are available in the FSLIST procedure, see *SAS Language Reference: Dictionary*.

External files (files maintained by the host operating system rather than by SAS software) can contain various types of information:

- data records
- output from previous SAS sessions
- SAS source statements
- carriage-control information.

Examples

- Browse an external file to which the fileref MYFILE has previously been assigned. The file contains FORTRAN-style carriage-control characters, and overprinting is allowed.

```
call fslist('myfile','fortcc ovp');
```

- Browse the external file named FSLIST.PUB. Double quotation marks surrounding the filename string indicate that a physical filename, not a fileref, is being passed to the FSLIST routine.

```
call fslist('"fslist.pub"');
```

Note: The form of the physical filename depends on the host operating system. △

See Also

"FILENAME" on page 403

FSVIEW

Displays a SAS table in tabular format

Category: SAS Table

Syntax

CALL FSVIEW(*table-name*<,*open-mode*
  <,*formula-entry*<,*options*>>>);

table-name
is the SAS table to be displayed. Use the format <*libref.*>*member-name*<(*data-set options*)>. If the libref is omitted, the default SAS data library, WORK, is assumed. You can add a list of SAS data set options in parentheses following the table name. All data set options are valid except FIRSTOBS= and OBS=. Refer to *SAS Language Reference: Dictionary* for a list of SAS data set options and their descriptions.
Type: Character

open-mode
specifies the type of access to the SAS table:

'ADD'
Add a new blank row to the table, and then open the FSVIEW window with the new row displayed for editing.

'BROWSE'
Open the FSVIEW window for reading rows. (This is the default.)

'EDIT'
Open the FSVIEW window for editing rows.

'NEW'
Open the FSVIEW NEW window for creating the specified SAS table as a new table, then open the FSVIEW window for entering values into the new table.
Type: Character

formula-entry
is the FORMULA catalog entry that defines custom features for the FSVIEW session or that controls the display and behavior of the session.

Specify this argument as <*libref.*>*catalog-name*<.*entry-name*<.FORMULA>>. A one- or two-level name is interpreted as a catalog name, and the default formula entry name is assumed. The default formula entry name is the same as the member name of the table specified in the *table-name* argument. (A one-level name is assumed to refer to a catalog in the default SAS data library, WORK.) If the specified catalog does not exist, it is created. If the specified formula entry does not already exist, a new formula entry is created.
Type: Character

options
specifies whether to disable certain FSVIEW window commands for the duration of the FSVIEW session. Separate multiple options with blanks.

'BRONLY'
disables the MODIFY command so that only browsing is allowed and **'EDIT'** and **'ADD'** modes are ignored.

'NOADD'
disables the ADD command so that new rows cannot be added to the table.

'NODELETE'
disables the DELETE command so that rows cannot be deleted.
Type: Character

Details The FSVIEW routine calls the FSVIEW procedure, which opens the FSVIEW window to display the specified SAS table. By default, the SAS table is opened for browsing. You can use the *open-mode* argument to specify that the table should instead be opened for editing. You can also specify a formula entry and other options for the FSVIEW session.

If you specify NEW for the *open-mode* argument, the FSVIEW NEW window is opened for the user to define a new SAS table before the FSVIEW window is opened to enter values for that table.

You can specify SAS data set options by enclosing them within parentheses immediately following the SAS table name in the *table-name* argument, as in the second example in the "Examples" section.

If you want to specify the *options* argument but do not want to specify a formula entry, then use a null string (`''`) for the *formula-name* argument.

To specify multiple values for the *options* argument, separate the values with blanks.

For more information about the commands that are available in the FSVIEW procedure, see *SAS Language Reference: Dictionary*.

Examples

- Browse a SAS table named PERSONAL in the default SAS data library WORK:

  ```
  call fsview('personal');
  ```

- Edit a SAS table named PERSONAL in the library MASTER. Only those rows in which the SITE column has the value 5 are displayed.

  ```
  call fsview
  ('master.personal(where=(site=5))','edit');
  ```

- Edit a SAS table named PERSONAL in the library MASTER. Rows cannot be added or deleted during the FSVIEW session.

  ```
  call fsview
  ('master.personal','edit','','noadd nodelete');
  ```

See Also

"FSEDIT" on page 433

"NEW" on page 561

FWRITE

Writes a record to an external file

Category: External File

Syntax

sysrc=**FWRITE**(*file-id*<,*cc*>);

sysrc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

file-id
contains the identifier that was assigned when the file was opened. If *file-id* is invalid, the program halts.
Type: Numeric

cc
specifies a carriage-control character:

'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.

Any other character (including a blank) starts the record on a new line.
If *cc* is not provided, FWRITE writes to the current line.
Type: Character

Details 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) with FOPEN.

Example

Write the numbers 1 to 50 to an external file, skipping two blank lines. Then call FSLIST to display the newly created file.

```
   /* Assign the fileref THEFILE to the external */
   /* filename stored in the variable FNAME.     */
rc=filename('thefile',fname);
fileid=fopen('thefile','o',0,'P');
do i=1 to 50;
   rc=fput(fileid,put(i,2.));

 if (fwrite(fileid,'-') ne 0) then do;
    _msg_=sysmsg();
    put msg;
    return;
  end;
end;

rc=fclose(fileid);
call fslist('thefile','cc');
```

See Also

"FAPPEND" on page 384

"FGET" on page 393

"FPUT" on page 427

GETFKEY

Returns the command that is assigned to a function key

Category: Keys

Syntax

key-command=**GETFKEY**(*key-name*);

key-command
returns the command that is currently assigned to the function key.
Type: Character

key-name
specifies the name of the function key as listed in the KEYS window. Function key names vary according to the device. Use FKEYNAME to retrieve the name of a function key.
Type: Character

Details GETFKEY returns the command that is assigned to a function key for the current window. This is the same as the text displayed for the key in the KEYS window.

You can use this function only in entries that have a DISPLAY window.

Examples

Example 1: Using GETFKEY When the Function Key Name Is Unknown Return the command assigned to the first function key if the name of the function key is not known:

```
command=getfkey(fkeyname(1));
```

Example 2: Using GETFKEY When the Function Key Name Is Known If the value of the first function key is F1, return the command that is assigned to the first function key:

```
command=getfkey('F1');
```

See Also

"FKEYNAME" on page 413

"NUMFKEYS" on page 574

"SETFKEY" on page 649

GETFOOT

Returns the text of a footnote definition

Category: SAS System Option

Syntax

foot-text=**GETFOOT**(*foot-num*);

foot-text
contains the text of the footnote definition, or a blank if the footnote is not defined.
Type: Character

foot-num
is the number (1 to 10) of the footnote definition.
Type: Numeric

Details Use GETFOOT to retrieve any footnote text that was previously defined in the SAS session by either the FOOTNOTE statement or the SCL SETFOOT routine. Only the footnote text is retrieved. Graphic options such as color or font are not returned.

You can view footnotes in the FOOTNOTES window by using the FOOTNOTE command. Changing any text in the FOOTNOTES window, however, resets all graphically defined FOOTNOTE options, such as color, font, and position.

For more information about footnotes, see *SAS Language Reference: Dictionary*. For more information about graphical footnotes, see *SAS/GRAPH Software: Reference*.

Example

Store the text of FOOTNOTE2 in the variable FNOTE2:

```
fnote2=getfoot(2);
```

See Also

"GETTITLE" on page 449

"SETFOOT" on page 650

"SETTITLE" on page 666

GETITEMC, GETITEML, GETITEMN, and GETITEMO

Returns a value that is identified by its position in an SCL list

Category: List

Syntax

cval=**GETITEMC**(*list-id*<,*index*>);

sublist-id=**GETITEML**(*list-id*<,*index*>);

nval=**GETITEMN**(*list-id*<,*index*>);

object-id=**GETITEMO**(*list-id*<,*index*>);

cval
contains the character value (returned by GETITEMC) of the item that is stored at the specified position in list identified by *list-id*.
Type: Character

sublist-id
contains the list identifier (returned by GETITEML) of the sublist that is stored at the specified position in the list identified by *list-id*.
Type: Numeric

nval
contains the numeric value (returned by GETITEMN) of the item that is stored at the specified position in the list identified by *list-id*.
Type: Numeric

object-id
contains the object identifier (returned by GETITEMO) of the object that is stored at the specified position in the list identified by *list-id*.
Type: Numeric

list-id
contains the identifier of the list that you want to query. An invalid *list-id* produces an error condition.
Type: Numeric or List

index
is the position in the list of the item to return. The position can be specified as a positive or negative number. By default, *index* is 1 (the first item). If *index* is a positive number, then the item is at position *index* from the beginning of the list. If *index* is a negative number, then the item is at position ABS(*index*) from the end of the list. An error condition results if the absolute value for *index* is zero or if it is greater than the number of items in the list.
Type: Numeric

Details An error results if you use any of these functions to return an indexed item that has a different data type. You can determine the data type of an item in a list by using ITEMTYPE before using GETITEMC, GETITEML, GETITEMN, or GETITEMO.

Examples

Example 1: Using the GETITEMC Function Return the character values stored in the first and third items of the list identified by the MYLIST variable:

```
citem=getitemc(mylist);
citem=getitemc(mylist,3);
```

Example 2: Using the GETITEML Function Return the list identifier stored in the eighth item of the list identified by the MYLIST variable:

```
slist=getiteml(mylist,8);
```

Example 3: Using the GETITEMN Function Return the numeric value stored in the fifth item of the list identified by the MYLIST variable:

```
nitem=getitemn(mylist,5);
```

Example 4: Using the GETITEMO Function Return the object identifiers stored in the third and ninth item of the list identified by the MYLIST variable:

```
sublist=getitemo(mylist,3);
oitem=getitemo(mylist,9);
```

See Also

"GETNITEMC, GETNITEML, GETNITEMN, and GETNITEMO" on page 445

"POPC, POPL, POPN, and POPO" on page 593

"SETITEMC, SETITEML, SETITEMN, and SETITEMO" on page 651

GETLATTR

Returns the attributes of either an SCL list or an item in the list

Category: List

Syntax

attributes=**GETLATTR**(*list-id*<,*index*>);

attributes
contains a string of words separated by blanks. Each word is a separate attribute for a list or item.
Type: Character

list-id
contains the identifier of the list that GETLATTR processes. An invalid *list-id* produces an error condition.
Type: Numeric or List

index
is the position of the item in the list. The position can be specified as a positive or negative number. By default, *index* is 1 (the first item). If *index* is a positive number, then the item is at position *index* from the beginning of the list. If *index* is a negative number, then the item is at position ABS(*index*) from the end of the list. An error condition results if the absolute value for *index* is zero or if it is greater than the number of items in the list.

If *index* is 0 or is omitted, then the attributes returned by GETLATTR are list attributes. If *index* is nonzero, then GETLATTR returns the attributes associated with the indexed item instead of the attributes for the entire list.
Type: Numeric

Details The items in *attributes* can be used to assign attributes to another list or item. The string returned as *attributes* contains a blank before and after each attribute, which makes it easy to determine whether an attribute is set by searching *attributes* for an attribute name. Use the INDEX function to search the string for a specified attribute.

If *index* is omitted, *attributes* contains one attribute from each row of the following table:

Default Setting	Alternate Setting
UPDATE	NOUPDATE
NOFIXEDTYPE	FIXEDTYPE
NOFIXEDLENGTH	FIXEDLENGTH
ANYNAMES	SASNAMES
DUPNAMES	NODUPNAMES
NOCHARONLY	CHARONLY
NONUMONLY	NUMONLY
COPY	NOCOPY
HONORCASE	NOHONORCASE

If *index* is supplied, then *attributes* is the set of item attributes consisting of one attribute from each row of the following table:

Default Setting	Alternate Setting
ACTIVE	INACTIVE
DELETE	NODELETE
NOFIXEDTYPE	FIXEDTYPE

Default Setting	**Alternate Setting**
UPDATE	NOUPDATE
WRITE	NOWRITE

For detailed information about these attributes, see “SETLATTR” on page 656.

Example

Create a list LISTID with one item and print the sets of list attributes for LISTID as well as the item attributes that are associated with the first item of LISTID. Note the leading and trailing blanks in the attribute strings, which are made evident by embedding the attribute strings in double quotation marks.

```
INIT:
   listid = makelist(1);
   listattrs = '"' || getlattr(listid) || '"';
   put listattrs=;
   found = index(listattrs,'UPDATE');
   put found=;
   itemattrs = '"' || getlattr(listid,1) || '"';
   put itemattrs=;
   rc = dellist(listid);
return;
```

The output of this example is

```
LISTATTRS=" DELETE UPDATE NOFIXEDTYPE
NOFIXEDLENGTH ANYNAMES DUPNAMES
NOCHARONLY NONUMONLY COPY NOHONORCASE"
FOUND=10;
ITEMATTRS=" ACTIVE WRITE NOAUTO NOEDIT
DELETE UPDATE NOFIXEDTYPE "
```

FOUND returns the starting position of the word "UPDATE" in the string of list attributes.

See Also

“HASATTR” on page 458

“SETLATTR” on page 656

GETNITEMC, GETNITEML, GETNITEMN, and GETNITEMO

Return a value identified by its item name in an SCL list

Category: List

Syntax

cval=**GETNITEMC**(*list-id*,*name*<,*occurrence*
<,*start-index*<,*default*<,*force*>>>>);

sublist-id=**GETNITEML**(*list-id*,*name*<,*occurrence*
<,*start-index*<,*default*<,*force*>>>>);

nval=**GETNITEMN**(*list-id*,*name*<,*occurrence*
<,*start-index*<,*default*<,*force*>>>>);

obj-val=**GETNITEMO**(*list-id*,*name*<,*occurrence*
<,*start-index*<,*default*<,*force*>>>>);

cval
contains the character value that is returned by GETNITEMC.
Type: Character

list-id
is the identifier of the list to search. An invalid *list-id* produces an error condition.
Type: List

name
is the name of the item to search in the list. Item names are converted to uppercase during the search if *force* is **'Y'** or if the searched list has the attribute NOHONORCASE set. Trailing blanks are ignored when the list is searched for a matching name. Thus, the names 'abc' and 'Abc' are converted to 'ABC'.
IGNORECASE is the alias for NOHONORCASE and is the default for a list. But you can use the SETLATTR function to set a list's attribute to HONORCASE.
Type: Character

occurrence
is the number of the occurrence of the named item to be searched. The default, 1, indicates the first occurrence of the item.
Type: Numeric or List

start-index
specifies where in the list to begin searching for the item. By default, *start-index* is 1 (the first item). If *start-index* is positive, the search begins at position *start-index* items from the beginning of the list. If *start-index* is negative, the search begins at the item specified by ABS(*start-index*) items from the end of the list. An error condition results if the absolute value of *start-index* is zero or if it is greater than the number of items in the list.
Type: Numeric

default
is a default value to return if the named item is not found in the list. This value must have the appropriate data type for the function that is being used:

GETNITEMC
a character value

GETNITEML
a sublist identifier or a list value

GETNITEMN
a numeric value

GETNITEMO
an object identifier.

Type: Character, Numeric, List, or Object

sublist-id
is the identifier for the sublist that is returned by GETNITEML.
Type: List

nval
is the numeric value that is returned by GETNITEMN.
Type: Numeric

obj-val
is the object identifier that is returned by GETNITEMO.
Type: Object

force
specifies whether to conduct the name search in uppercase.

'N'	searches according to the list attributes HONORCASE and NOHONORCASE, which are specified with SETLATTR. (This is the default.)
'Y'	conducts the name search in uppercase regardless of list attributes specified with SETLATTR.

Type: Character

Details By default, the search starts at the beginning of the list and returns the first item found that has the specified item name. However, you can start the search at a different place in the list by specifying a *start-index* other than 1. You can also specify a different occurrence of the item (for example, the second, tenth, or twentieth) by specifying an *occurrence* other than 1. If the item is not found and you have specified a value for *default* as the fifth parameter, then that value is returned instead of an error condition.

If *occurrence* and *start-index* are both positive or both negative, the search proceeds forward from the *start-index* item. For forward searches, the search continues only to the end of the list and does not wrap back to the front of the list. If *occurrence* or *start-index* is negative, then the search is backwards. For backward searches, the search continues only to the beginning of the list and does not wrap back to the end of the list.

GETNITEMC combines the actions of NAMEDITEM and GETITEMC. GETNITEML combines the actions of NAMEDITEM and GETITEML. GETNITEMN combines the actions of NAMEDITEM and GETITEMN. GETNITEMO combines the actions of NAMEDITEM and GETITEMO.

In situations where your application manipulates an SCL list and you cannot guarantee that the named item is character, you should not use GETNITEMC. Instead, when manipulating SCL lists which may contain other types, you should use NAMEDITEM with ITEMTYPE with GETITEMC, GETITEML, GETITEMN, or GETITEMO.

An error condition results if

- the named item is not a character value and you are using GETNITEMC
- the item is not a list identifier and you are using GETNITEML
- the item is not a numeric value and you are using GETNITEMN
- the item is not an object and you are using GETNITEMO.
- the named item does not exist and *default* is not specified.

Examples

Example 1: Using the GETNITEMC Function Halt the program if there are fewer than two items named **'Software Sales'** in the list identified by DIRECTORY. Omitting the default value from GETNITEMC designates that the character item must exist in the list.

```
s=getnitemc(directory,'Software Sales',2,-1);
```

This statement is equivalent to the following statements:

```
ssi=nameditem(directory,'Software Sales',2,-1);
s=getitemc(directory,ssi);
```

Both of the preceding examples search for the second occurrence of **'Software Sales'**, starting from the end of the list.

Example 2: Searching for an Item of Unknown Type This example shows how to search for a named item in an SCL list when you do not know the type of that item.

```
index = nameditem(listid, 'A', occurrence,
                               startIndex);
if index then
   select (itemtype(listid, index));
     when ('C') c = getitemc(listid, index);
     when ('L') l = getiteml(listid, index);
     when ('N') n = getitemn(listid, index);
     when ('O') o = getitemo(listid, index);
   end;
```

Example 3: Using GETNITEML and Specifying a Default Value If the named item may not be in the list, supply a list identifier value for *default* for GETNITEML:

```
sslistid=getniteml
(emp_list,'Marketing',2,-10,-1);
```

The preceding program statement is equivalent to the following:

```
mpos=nameditem(emp_list,'Marketing',2,-10);
if mpos ne 0 then
   sslistid=getiteml(emp_list,mpos);
else
   sslistid=-1;
```

Example 4: Using GETNITEMC and Specifying a Default Value This example shows GETNITEMC, using a default value that contains an error message.

```
defaultc='Value not found';
s=getnitemc
(directory,'Software Sales',2,-1,defaultc);
```

See Also

"GETITEMC, GETITEML, GETITEMN, and GETITEMO" on page 442

"NAMEDITEM" on page 555

"NAMEITEM" on page 558

"SEARCHC, SEARCHL, SEARCHN, and SEARCHO" on page 636

"SETLATTR" on page 656

"SETNITEMC, SETNITEML, SETNITEMN, and SETNITEMO" on page 661

GETPARMID

Returns the numeric value stored by the SETPARMID routine

Category: Modular Programming and Object Oriented

Syntax

nval=**GETPARMID**();

nval
contains the numeric value stored by a previous call to the SETPARMID routine.
Type: Numeric

Details SETPARMID stores a value, and GETPARMID retrieves the stored value. SETPARMID and GETPARMID allow only one value to be passed. To pass multiple values between entries, use the ENTRY statement. Additional ways of making values available to other SCL programs include using macro variables and SCL lists.

Example

Retrieve the table identifier value that was stored in another program by SETPARMID:

```
dsid=getparmid();
```

See Also

"DISPLAY" on page 350

"ENTRY" on page 369

"SETPARMID" on page 664

GETTITLE

Returns the text of a title definition

Category: SAS System Option

Syntax

title-text=**GETTITLE**(*title-num*);

title-text
is the text of the title definition, or a blank if the title is not defined.
Type: Character

title-num
is the number (1 to 10) of the title definition.
Type: Numeric

Details Use GETTITLE to retrieve any title text that was previously defined in the SAS session by either the TITLE statement or the SCL SETTITLE routine. Only the title text is retrieved. Graphic options, such as color or font, are not returned.

You can view titles in the TITLES window by using the TITLE command. Changing any text in the TITLES window, however, resets all graphically defined title options, such as color, font, and position.

For more information about titles, see *SAS Language Reference: Dictionary*. For more information about graphical titles, see *SAS/GRAPH Software: Reference*.

Example

Put the text of TITLE2 into the variable TITLE2:

```
title2=gettitle(2);
```

See Also

"GETFOOT" on page 441

"SETFOOT" on page 650

"SETTITLE" on page 666

GETVARC and GETVARN

Assign the value of a SAS table column to an SCL variable

Category: SAS Table

Syntax

cval=**GETVARC**(*table-id*,*col-num*);

nval=**GETVARN**(*table-id*,*col-num*);

cval
is the value of the character column that is returned by GETVARC.
Type: Character

table-id
is the identifier for a table that is open. If *table-id* is invalid, the program halts.
Type: Numeric

col-num
is the number of the column in the Table Data Vector (TDV). This value can be obtained by using the VARNUM function. If the column specified in *col-num* is invalid, the program halts.
Type: Numeric

nval
is the value of the numeric column that is returned by GETVARN.
Type: Numeric

Details Before you use GETVARC or GETVARN, you can use VARNUM to obtain the number of a column in a SAS table. You can nest VARNUM, or you can assign it to a column that can be passed as the second argument. GETVARC and GETVARN read the value of the specified column that is in the current row in the TDV and copy that value to the specified SCL variable in the SCL data vector (SDV).

Examples

Example 1: Using GETVARN to Search for a Value Assign VARNUM to a column that can be passed as the second argument to GETVARN. Read row number 10 into the TDV.

```
pricenum=varnum(mydataid,'price');
rc=fetchobs(mydataid,10);
price=getvarn(mydataid,pricenum);
```

Example 2: Using GETVARC with a Nested VARNUM Nest VARNUM in the GETVARC function to search for the value of the character column NAME from the tenth row of the open SAS table whose identifier is stored in the column MYDATAID.

```
rc=fetchobs(mydataid,10);
user=getvarc(mydataid,varnum(mydataid,'name'));
```

See Also

"FETCH" on page 389

"FETCHOBS" on page 390

"LOCATEC and LOCATEN" on page 527

"PUTVARC and PUTVARN" on page 609

"UPDATE" on page 697

"VARNUM" on page 713

GETVARF

Assigns the formatted value of a SAS table column to a character SCL variable

Category: SAS Table

Syntax

cval=**GETVARF**(*table-id*,*col-num*);

cval
is the formatted value of the table column that is returned by GETVARF.
Type: Character

table-id
is the identifier for a table that is open. If *table-id* is invalid, the program halts.
Type: Numeric

col-num
is the number of the column in the Table Data Vector (TDV). This value can be obtained by using the VARNUM function. If the column specified in *col-num* is invalid, the program halts.
Type: Numeric

Details GETVARF assigns the formatted value of a SAS table column to a character SCL variable. If no format has been assigned to the specified column, GETVARF returns the raw value for a character column or _BLANK_ for a numeric column.

Examples

Example 1: Print Formatted Table Column Values The example first creates a SAS table with a character column, NAME, and two numeric columns, BDAY and GENDER. It then reads each row from the table and prints the values of each column plus the formatted values of the two numeric columns.

```
control asis;
submit continue;
  proc format; value sexfmt 1='Male' 2='Female';
  data work.samplef; input name $ 1-10  bday date. gender;
  format  bday date. sex sexfmt.  ;
  cards;
Jane      16oct63  2
Bill      15may62  1
Mary      25jan64  2
;
endsubmit;

id = open ( 'work.samplef');
do while (fetch(id)=0 );
  name  = getvarc ( id, 1);
  bdayn = getvarn (id, 2);
  bday  = getvarf (id, 2);
  gender  = getvarn (id, 3);
  fgender = getvarf (id, 3);
  put name= bdayn= bday= gendern= gender=;
  end;
rc = close (id);
```

The output would be like the following:

```
name=Jane bdayn=1384 bday= 16OCT63 gendern=2 gender=Female
name=Bill bdayn=865 bday= 15MAY62 gendern=1 gender=Male
name=Mary bdayn=1485 bday= 25JAN64 gendern=2 gender=Female
```

See Also

"GETVARC and GETVARN" on page 450

"GETVARC and GETVARN" on page 450

GGLOBAL

Returns the text of a SYMBOL, PATTERN, LEGEND, or AXIS statement

Category: SAS System Option

Syntax

stmt-text=**GGLOBAL**(*stmt-type*,*stmt-num*);

stmt-text
contains the text of the retrieved SYMBOL, PATTERN, LEGEND, or AXIS statement. If *stmt-type* is invalid, a missing value is returned.
Type: Character

stmt-type
is the type of statement to retrieve:

'SYMBOL'

'PATTERN'

'LEGEND'

'AXIS'

Type: Character

stmt-num
is the number of the SYMBOL, PATTERN, LEGEND, or AXIS statement to retrieve. Valid values are from 1 to the number of statements that are defined for the specified type, which is returned by the GGLOBALN function.
Type: Numeric

Details Because a user can change SYMBOL, PATTERN, LEGEND, or AXIS statements during the execution of an application, GGLOBALN must be executed before the GGLOBAL function in order to set up an internal table that is used by GGLOBAL.

Note: SYMBOL and PATTERN can generate more than one definition per statement. For more information about the SYMBOL, PATTERN, LEGEND, and AXIS statements, see *SAS/GRAPH Software: Reference*. △

Example

Assume that the following SYMBOL statements have been defined for the current SAS session:

```
symbol1 c=red;
symbol30 c=blue;
```

Check to see that at least two SYMBOL statements are available. If this condition is true, the text of the second SYMBOL statement is returned to the variable SYMBOL2.

```
numsymb=ggloBaln('symbol');
if (numsymb >= 2)
then symbol2=gglobal('symbol',2);
```

The value returned to NUMSYMB is 2. The following value is returned to SYMBOL2:

```
SYMBOL30 CV=BLUE CO=BLUE CI=BLUE;
```

The value of HEIGHT is also returned:

```
SYMBOL30 CV=BLUE CO=BLUE CI=BLUE HEIGHT=1 ;
```

See Also

"GGLOBALE" on page 454

"GGLOBALN" on page 454

GGLOBALE

Deletes an internal table of SYMBOL, PATTERN, LEGEND, or AXIS definitions

Category: SAS System Option

Syntax

sysrc=**GGLOBALE**(*stmt-type*);

sysrc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

stmt-type
specifies the type of statement to delete:

'SYMBOL'

'PATTERN'

'LEGEND'

'AXIS'

Type: Character

Details When you have completed processing information concerning the SYMBOL, PATTERN, LEGEND, or AXIS statements, use GGLOBALE to free the memory used for storing the internal table that was created with GGLOBALN. For more information about the SYMBOL, PATTERN, LEGEND, and AXIS statements, see *SAS/GRAPH Software: Reference*.

Example

Free the internal table that was created by GGLOBALN for the SYMBOL statements, and check the return code to determine whether a message needs to be issued:

```
rc=gglobale('symbol');
if rc then _msg_=sysmsg();
```

See Also

"GGLOBAL" on page 452

"GGLOBALN" on page 454

GGLOBALN

Returns the number of SYMBOL, PATTERN, LEGEND, or AXIS statements that are currently defined

Category: SAS System Option

Syntax

num-stmts=**GGLOBALN**(*stmt-type*);

num-stmts
contains the number of SYMBOL, PATTERN, LEGEND, or AXIS definitions that are currently defined.
Type: Numeric

stmt-type
specifies the type of statement to return:

'SYMBOL'

'PATTERN'

'LEGEND'

'AXIS'

Type: Character

Details Information about SYMBOL, PATTERN, LEGEND, or AXIS statements is stored in an internal table and can be retrieved with GGLOBAL. To delete the internal table created by GGLOBALN, use GGLOBALE.

Note: SYMBOL and PATTERN can generate more than one definition per statement. For more information about SYMBOL, PATTERN, LEGEND, and AXIS statements, see *SAS/GRAPH Software: Reference*. △

Example

Assume that the following SYMBOL statements have been defined for the current SAS session:

```
symbol1 c=red;
symbol30 c=blue;
```

In the variable NUMSYMB, return the number of SYMBOL statements that are currently available. The value returned for NUMSYMB is 2, not 30.

```
numsymb=ggloballn('symbol');
```

See Also

"GGLOBAL" on page 452

"GGLOBALE" on page 454

GOTO

Branches immediately to another entry

Category: Control Flow

Syntax

CALL GOTO(*entry*<,*action*<,*frame*>>);

entry
is the name of the entry to branch to. The entry can be any of the SAS/AF display entry types (FRAME, PROGRAM, MENU, CBT, or HELP). The *entry* argument can be in the following forms:

entry
to specify a PROGRAM entry in the current catalog.

entry.type
to specify an entry in the current catalog.

libref.catalog.entry
to specify a PROGRAM entry in a different catalog.

libref.cat-name.entry.type
to specify an entry in a different catalog.
Type: Character

action
specifies how the execution stack is to be handled and where control transfers to when the specified entry ends:

'A'	adds *entry* to the top of the execution stack. The specified entry is displayed immediately. When the entry ends, the user returns to the window that was displayed before the program with the CALL GOTO was executed.
'C'	clears the current execution stack. The specified entry is displayed immediately, and the stack is cleared. When the entry ends, the user returns to the parent entry, if one was specified in the entry, or exits the AF window. This option may be useful if you have memory constraints. (This is the default.)
'R'	removes the top entry from the execution stack and places the entry specified in the GOTO routine on the top of the execution stack. The specified entry is displayed immediately. When the entry ends, the user returns to the next entry on the stack rather than to the program that contains the GOTO call.

Type: Character

frame
is the number of the CBT frame if you are branching to a CBT entry.
Type: Numeric

Details The GOTO routine branches immediately to a CBT, HELP, MENU, FRAME, or PROGRAM entry and transfers control to it. Statements that appear after GOTO are not executed, because control is transferred to the entry that is specified in the GOTO routine.

GOTO, which always starts a new stream, cannot be used in FSEDIT or FSVIEW programs.

Example

Pass control to MYEND.PROGRAM, and end the SAS/AF session if the user issues the END command. Assume there is no parent entry specified.

```
if _status_='E'
then call goto('myend.program','C');
```

See Also

"DISPLAY" on page 350

GRAY

Grays FRAME entry controls and stations of a choice group

Category: Control or Field

Syntax

rc=**GRAY**(*var-name*<,*station*<,*row*>>);

rc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

var-name
is the name of a window control or choice group to be grayed.
Type: Character

station
is the number of the button in a radio box or field in a choice group. This value must be greater than 0 and no larger than the total number of stations defined for the choice group. For PROGRAM entries, use the value 0 as a placeholder if the entire choice group at a specified row is to be grayed.
Type: Numeric

row
is the number of the row when the choice group is in the scrollable section of an extended table. *Row* can be used in PROGRAM entry programs but not for FRAME entry programs. Specify *row* only when you want to gray a station from outside an extended table's getrow or putrow section. Do not specify *row* if you want to gray a station from a getrow or putrow section.
Type: Numeric

Details Users cannot select a FRAME entry control, choice group, or choice group station that is grayed.

For linked action fields in choice groups, the ACTION-type pair is considered one station. For example, the following line in a PROGRAM entry window defines three linked action fields:

```
& &A______     & &B______     & &C______
```

To gray the linked action pair for field B, pass in 2 for the value of *station*, not 4.

Window controls can also use the _gray method. You cannot use GRAY in FSEDIT or FSVIEW programs.

Example

Prevent users from selecting CONTINUE when the value of AGE is less than 21:

```
if (age<21) then
      rc=gray('CONTINUE');
   else
      rc=ungray('CONTINUE');
```

See Also

"ACTIVATE" on page 249

"ISACTIVE" on page 492

"ISGRAY" on page 493

"UNGRAY" on page 693

HASATTR

Reports whether an SCL list or a list item has a specified attribute

Category: List

Syntax

rc=**HASATTR**(*list-id*,*attribute*<,*index*>);

rc
contains the return code for the operation:

1	The list or item has the specified attribute.
0	The list or item does not have the specified attribute.

Type: Numeric

list-id
is the identifier of the list that HASATTR searches. An invalid *list-id* produces an error condition.
Type: Numeric or List

attribute
is an attribute for a list or list item, as described in SETLATTR. In addition, you can test for the following special attributes:

'G'	returns 1 if the list is a global list.
'L'	returns 1 if the list is a local list.
'C'	returns 1 if the list identifier is also a class identifier.

`'O'` returns 1 if the list identifier is also an object identifier.

Type: Character

index

is the position of the item in the list. The position can be specified as a positive or negative number. By default, *index* is 1 (the first item). If *index* is omitted, HASATTR checks to see if the specified list has the named *attribute*. If *index* is specified, HASATTR checks to see if the specified item has the named *attribute*. If *index* is a positive number, then the item is at position *index* from the beginning of the list. If *index* is a negative number, then the item is at position ABS(*index*) from the end of the list. An error condition results if the absolute value for *index* is zero or if it is greater than the number of items in the list.

Type: Numeric

Details If no value is specified for *index*, HASATTR queries the attribute of the list. If a nonzero value is specified for *index*, HASATTR queries the attribute of an item in the list.

For a list of attributes for lists and list items, see "SETLATTR" on page 656.

Examples

Example 1: Clearing a List That Has a Particular Attribute Clear the list identified by MYLIST only if it has the UPDATE attribute:

```
if hasattr(mylist,'UPDATE') then rc=clearlist
                                  (mylist);
```

Example 2: Determining Whether a List Item Has a Particular Attribute Determine whether the third item in a list has the FIXEDTYPE attribute:

```
isfixed=hasattr(mylist,'FIXEDTYPE',3);
```

See Also

"GETLATTR" on page 443

"SETLATTR" on page 656

HOME

Positions the cursor on a window's command area

Category: Control or Field

Syntax

HOME;

Details In SAS/AF applications, the HOME statement moves the cursor immediately. In FSEDIT and FSVIEW applications, the cursor is moved when control is returned to the application.

See "CURSOR" on page 323 for more information about how to position the cursor in a field.

If the PMENU facility is active, the command area for a dialog window is at the bottom of the window.

Some systems do not position the cursor on the pull-down menu for standard windows nor on the command area for dialog windows.

See Also

"CURSOR" on page 323

"FIELD" on page 394

ICCREATE

Creates integrity constraints on a SAS table

Category: SAS Table

Syntax

sysrc=**ICCREATE**(*table-id*,*icname*,*ictype*,*icvalue*<, *argument-1*<, . . . <,*argument-3*>>>);

sysrc
contains the status of the operation:

=0	successful
>0	not successful
<0	the operation was completed with a warning

Type: Numeric

table-id
contains the identifier for the SAS table, which is returned by the OPEN function. Integrity constraints can be set for a new SAS table only when it is opened in NEW mode. Integrity constraints can be set for an existing table only when it is opened in UTILITY or NEW mode. (Remember that if you open an existing table in NEW mode, the table is replaced without warning.)

Type: Numeric

icname
is the name of the integrity constraint.

Type: Character

ictype
specifies the type of integrity constraint to create:

'CHECK'
specifies that column values must fall within a particular set, range, or list of values, or that values must meet a condition or be the same as values in other columns in the same row.

'FOREIGN'
specifies that a column is linked to a column in another SAS table. A foreign key column can contain only a null value or a value that is present in the associated primary key column.

'NOT-NULL'
specifies that column values cannot contain missing values.

'PRIMARY'
specifies that column values must be unique and that they cannot be missing values. If there is an associated foreign key column, then primary key values cannot be deleted or changed unless the same deletions or changes have been made in values of that foreign key column.

'UNIQUE' | **'DISTINCT'**
specifies that column values must be unique. If more than one column is specified, then their concatenated values must be unique.
Type: Character

icvalue
specifies the names of one or more columns from the SAS table to which the constraint is applied. Separate multiple column names with blanks. If *ictype* is **CHECK**, then *icvalue* can contain a condition that values must meet.
Type: Character

argument-1,...,argument-3
are additional specifications that are used when *ictype* is **FOREIGN**.

argument-1
is the name of the SAS table in which one or more associated primary keys are defined. This links the two SAS tables and creates the primary/foreign key relationship. Use *libref.member* to specify the SAS table. This argument is required for foreign keys.

argument-2
is the restriction option for update operations on values in foreign key columns:

'RESTRICT'
allows the primary key value to be updated only if no foreign key matches the current value to be updated.

'NULL'
specifies that when the primary key value is modified, all matching foreign key values are set to NULL.

argument-3
is the restriction option for delete operations in foreign key columns:

'RESTRICT'
allows the primary key row to be deleted only if no foreign key values match the deleted value.

'NULL'
allows the primary key row to be deleted and sets the values of corresponding foreign keys to null.
Type: Character

Details ICCREATE defines integrity constraints for a SAS table that has been opened in an SCL program. Integrity constraints guarantee the correctness and consistency of data that is stored in a SAS table. Integrity constraints are enforced automatically for each addition, update, or deletion of data in a table that contains a constraint.

You can define integrity constraints for SAS tables that contain zero or more rows. For tables that already contain data, an integrity constraint is compared to values in all table rows. If a single row does not comply with the constraint being defined, then the creation of the constraint fails. When rows are added to tables that have integrity constraints, the new rows are checked against the constraints. If a row violates an integrity constraint, the row is not added to the table. Also, a primary key column cannot be dropped until all foreign key columns that reference the primary key are deleted.

The basic types of integrity constraints are general constraints and referential constraints. The general constraints, which control values in a single SAS table, are CHECK, NOT-NULL, PRIMARY key, and UNIQUE. Referential constraints, which establish a parent-child relationship between columns in two SAS tables, include FOREIGN keys and PRIMARY keys that have one or more FOREIGN key references. A FOREIGN key column (the child) can contain only null values or values that are present in its associated PRIMARY key (the parent). Values for a FOREIGN key cannot be added unless the same values also exist in the associated PRIMARY key.

For more information about integrity constraints, see “Preserving the Integrity of Data” on page 177.

Example

This example creates integrity constraints for the SAS tables MYLIB.ONE and MYLIB.TWO. Although they contain different data, they have one column with shared data, an identifier number that is stored in IDNUM in MYLIB.ONE and in EMPID in MYLIB.TWO. The following *icname*s are used in the example:

UQ	is a UNIQUE constraint which specifies that the concatenated values of columns D and E must be unique.
WH	is a CHECK constraint which specifies that the sum of the values of columns B and C must be less than 1000.
PK	is a PRIMARY constraint which specifies that the IDNUM column can contain only unique and non-missing values. Because IDNUM is associated with the foreign key column EMPID in the table MYLIB.TWO, values for IDNUM cannot be deleted or changed unless the same changes have been made to the values in the foreign key EMPID.
FK	is a foreign key in the table MYLIB.TWO. EMPID is the foreign key column whose primary key is IDNUM in MYLIB.ONE. Because EMPID is a foreign key column, it can contain only values that are present in IDNUM. The first **RESTRICT** specifies that a value in IDNUM can be changed only if EMPID does not contain a value that matches the IDNUM value to be changed. The second **RESTRICT** specifies that a row can be deleted from MYLIB.ONE only if the value of IDNUM does not match a value in EMPID in MYLIB.TWO.
NONULL	is a NOT-NULL constraint which specifies that the EMPID column cannot contain a null value.

```
table1=open('mylib.one','V');
rc=iccreate(table1,'uq','Unique','d e');
rc=iccreate(table1,'wh','Check','(b + c)< 1000');
rc=iccreate(table1,'pk','Primary','idnum');
rc=close(table1);

table2=open('mylib.two','V');
rc=iccreate(table2,'fk','Foreign','empid',
   'mylib.one','restrict','restrict');
```

```
rc=iccreate(table2,'nonull','Not-null','empid');
    ...more SCL statements...
rc=close(table2);
```

See Also

"ICDELETE" on page 463

"ICTYPE" on page 466

"ICVALUE" on page 467

ICDELETE

Drops an integrity constraint from a SAS table

Category: SAS Table

Syntax

sysrc=**ICDELETE**(*table-id*,*icname*);

sysrc
contains the status of the operation:

=0	successful
>0	not successful
<0	the operation was completed with a warning

Type: Numeric

table-id
contains the identifier for the SAS table, which is returned by the OPEN function. An integrity constraint can be dropped only from a SAS table that is open in NEW or UTILITY mode.
Type: Numeric

icname
is the name of the integrity constraint to delete.
Type: Character

Details Referential integrity constraints (where *ictype* is **FOREIGN**) provide a link between SAS tables. Part of the link mechanism involves the name of the table. Therefore, renaming or deleting a table that has a FOREIGN integrity constraint is not allowed until after the FOREIGN key is deleted.

Example

Delete the integrity constraint UQ from the table MYLIB.ONE. The constraint was created in "Example" on page 462.

```
table1=open('mylib.one','V');
rc=icdelete(table1,'uq');
rc=close(table1);
```

See Also

"ICCREATE" on page 460

"ICTYPE" on page 466

"ICVALUE" on page 467

ICON

Associates an icon with a window

Category: Window

Syntax

rc=**ICON**(*icon-number*);

rc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

icon-number
is the number of the icon to use to represent the application window when it is minimized. If you specify a number for which no icon is defined, the SAS icon is used.
Type: Numeric

Details When a user minimizes a window into an icon, SAS uses the specified icon to represent the window.
Only systems that display icons support this function. Non-graphical devices ignore the icon setting.
The ICON function is ignored if you are running with the SAS software Application Work Space (AWS). To run an application without the AWS, specify the AWS=NO option in the AF command.

Example

Assign icon number 107 as the icon for the current window:

```
rc=icon(107);
```

ICREATE

Creates an index for a SAS table

Category: SAS Table

Syntax

sysrc=**ICREATE**(*table-id*,*key-name*<,*var-list*<,*options*>>);

sysrc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

table-id
is the identifier that was assigned when the table was opened. If *table-id* is invalid or if the table is not opened in UTILITY mode, the index is not created and the function returns a nonzero value.
Type: Numeric

key-name
is the name of the index key to be created.
Type: Character

var-list
is one or more columns from the SAS table to be indexed. Separate multiple names with blanks.
Type: Character

options
are index attributes, with multiple values separated by blanks within a single set of parentheses:

'NONUNIQUE' | **'UNIQUE'**
specifies whether the values of the key columns must be unique. The default is **'NONUNIQUE'**.

'MISSING' | **'NOMISS'**
specifies whether the index can point to missing values. The default is **'MISSING'**.
Type: Character

Details An *index* is an auxiliary data structure used to speed up searches for records that are specified by the value of a column (for example, "all the records for which AGE is greater than 65"). To create an index for a SAS table, you must open the table in UTILITY mode (see “OPEN” on page 576 for details).

An index on a single column is called a *simple index*. If *var-list* contains only one column name, then a simple index is created.

Note: For a simple index, *key-name* and *var-list* must both contain the same value. If *var-list* is omitted, then *key-name* specifies the index column. △

An index on more than one column is called a *composite index*. If *var-list* contains more than one column name, a composite index is created. In this case, *key-name* can be any valid SAS name that is not already used as a column name in the table. A composite index is based on the values of these columns, concatenated to form a single value.

UNIQUE specifies that the index contains only unique values of the key columns. The creation of such an index prohibits duplicate values for its columns from being stored in the SAS table. For columns that must be uniquely specified in a row, such as passport numbers, this option is useful for preventing duplicate values from incorrectly getting into a table. The function returns an error condition if non-unique values are present and UNIQUE is specified. By default, duplicate values are permitted in an index and thus in its table.

NOMISS prevents missing values from being pointed to by an index. Unlike UNIQUE, NOMISS does not prevent missing values from being stored in the SAS table. This feature is useful if the key columns contain many missing values that would make the index large and thus slower to access than if they were excluded. By default, missing values are stored in the index.

Indexes can also be created using

- the DATASETS and SQL procedures in base SAS software
- SAS/IML software
- the CONTENTS function in SCL
- the ACCESS procedure in SAS/ACCESS software.

Example

Create a simple index for the SAS table WORK.INVOICE. The key column for the index is the table column ITEMTYPE.

```
tableid=open('work.invoice','v');
   /* open in UTILITY mode */
rc=icreate(tableid,'itemtype',' ','unique nomiss');
```

In this example, because the value of the *var-list* argument is blank, the key column for the index is the column named in the *key-name* argument.

See Also

"CONTENTS" on page 299

"IDELETE" on page 468

"IOPTION" on page 490

"ISINDEX" on page 494

"IVARLIST" on page 498

"OPEN" on page 576

ICTYPE

Returns the type of integrity constraint that is assigned to a SAS table

Category: SAS Table

Syntax

ictype=**ICTYPE**(*table-id*,*icname*);

ictype
contains the type of integrity constraint that is returned. The constraints are listed below and are defined in “ICCREATE” on page 460:

'CHECK'

'FOREIGN'

'NOT-NULL'

'PRIMARY'

'UNIQUE'

Type: Character

table-id
contains the identifier for the SAS table, which is returned by the OPEN function.
Type: Numeric

icname
is the name of the integrity constraint.
Type: Character

Details ICTYPE returns the type of integrity constraint for a SAS table when you specify the name of the constraint.

Example

Return the type of the UQ integrity constraint that is assigned to the SAS table MYLIB.ONE. Because the constraint UQ (which was created in “ICCREATE” on page 460)was UNIQUE, the value of TYPE will be **UNIQUE**.

```
tableid=open('mylib.one','i');
type=ictype(tableid,'uq');
   ...more SCL statements...
rc=close(tableid);
```

See Also

“ICCREATE” on page 460

“ICDELETE” on page 463

“ICVALUE” on page 467

ICVALUE

Returns the column names or the condition associated with an integrity constraint

Category: SAS Table

Syntax

icvalue=**ICVALUE**(*table-id*,*icname*);

icvalue
contains the names of one or more columns from the SAS table identified by *table-id*. When *ictype* is **CHECK**, *icvalue* returns the condition that values must meet.
Type: Character

table-id
contains the identifier for the SAS table, which is returned by the OPEN function.
Type: Numeric

icname
is the name of the integrity constraint.
Type: Character

Details ICVALUE returns the names of columns that are associated with the specified integrity constraint. If *ictype* is **CHECK** and is specified as a condition, then ICVALUE returns the condition that is assigned to the constraint.

Example

Return the value of the UQ integrity constraint. If ICVALUE returns a blank, then display the error message. Because the constraint UQ (which was created in "ICCREATE" on page 460)specified that columns D and E must contain unique values, COLLIST would contain **D E**.

```
tableid=open('mylib.one','i');
collist=icvalue(tableid,'uq');
if (collist=' ') then _msg_=sysmsg();
```

See Also

"ICCREATE" on page 460

"ICDELETE" on page 463

"ICTYPE" on page 466

IDELETE

Deletes an index from a SAS table

Category: SAS Table

Syntax

sysrc=**IDELETE**(*table-id*,*key-name*);

sysrc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

table-id
is the identifier that was assigned when the table was opened. If *table-id* is invalid, the program halts.
Type: Numeric

key-name
is the name of the index key to be deleted.
Type: Character

Details In order to delete an index for a SAS table, you must open the table in UTILITY mode (see “OPEN” on page 576 for details).

You can also delete indexes using

- the DATASETS and SQL procedures in base SAS software
- the CONTENTS function in SCL
- the ACCESS procedure in SAS/ACCESS software.

Example

Delete an index for the SAS table WORK.INVOICE. The name of the index key is ITEMTYPE.

```
tableid=open('work.invoice','v');
rc=idelete(tableid,'itemtype');
```

See Also

“CONTENTS” on page 299

“ICREATE” on page 464

“IOPTION” on page 490

“ISINDEX” on page 494

“IVARLIST” on page 498

IMGCTRL

Performs control operations on the Image window

Category: Image

Syntax

rc=**IMGCTRL**(*task-id*,*command*,<*other-arguments*>);

rc
contains the return code for the operation:

0	successful
>0	not successful

Type: Numeric

task-id
contains the identifier of the image task, which is returned by a previous IMGINIT function.
Type: Numeric

command
is the control command to be executed. Commands are described in "Commands Used with IMGCTRL" on page 470.
Type: Character

other-arguments
lists the argument(s) required with each command.
Type: Character or Numeric

Commands Used with IMGCTRL

'WAIT' *seconds*
specifies the amount of time to wait before executing the next command. *Seconds* specifies the number of seconds to wait.

'WRAISE'
attempts to force the Image window to the top while IMGOP or IMGCTRL commands are executing. This command may not be honored by some window managers. Note that the 'TOPWINDOW' option can be set at IMGINIT time to force the window to always be on top.

'WSIZE' *width, height <xpos, ypos>*
specifies the width and height of the Image window in pixels. It optionally positions the window at *xpos* and *ypos* of the top left corner. Some window managers may not support positioning.

'WTITLE' *title'*
specifies a title for the Image window. The specified title appears in parentheses after 'SAS: IMAGE'.

Details IMGCTRL enables you to control the Image window. It includes, for example, commands to assign a window title and set the window size.

Examples

Example 1: Using the WTITLE Command Extract the name of the IMAGE entry and then use the WTITLE command to assign that name to the window:

```
path=lnamemk(5,catname,'format=cat');
rc=lnameget(path,type,name,form);
gname=scan(name,3,'.');
rc=imgop(taskid,'READ',path);
rc=imgctrl(taskid,'WTITLE',gname);
rc=imgop(taskid,'PASTE',1,1);
```

Example 2: Using the WAIT Command Wait 5 seconds before displaying the image after each PASTE command:

```
rc=imgop(taskid,'READ',path);
rc=imgop(taskid,'PASTE');
rc=imgctrl(taskid,'WAIT',5);
rc=imgop(taskid,'READ',path2);
rc=imgop(taskid,'PASTE');
rc=imgctrl(taskid,'WAIT',5);
```

Example 3: Using the WRAISE Command Force the Image window to the top:

```
pop:
   rc=imgctrl(taskid,'WRAISE');
```

```
return;
```

Example 4: Using the WSIZE Command Make the Image window match the size of the image that is being displayed:

```
height=0;
width=0;
rc=imgop(taskid,'READ',path);
rc=imgop(taskid,'QUERYN','WIDTH',iwidth);
rc=imgop(taskid,'QUERYN','HEIGHT',iheight);
rc=imgctrl(taskid,'WSIZE',iwidth,iheight);
rc=imgop(taskid,'PASTE',1,1);
```

Example 5: Using the WTITLE and WRAISE Commands Change the window title and then force the Image window to the top:

```
path=lnamemk(5,catname,'format=cat');
rc=lnameget(path,type,name,form);
gname=scan(name,3,'.');
rc=imgop(taskid,'READ',path);
rc=imgctrl(taskid,'WTITLE',gname);
rc=imgop(taskid,'PASTE',1,1);
rc=imgctrl(taskid,'WRAISE');
```

IMGINIT

Starts an image task

Category: Image

Syntax

task-id=**IMGINIT**(<*list-id*>,<*option*>);

task-id
contains the identifier for the image task. This identifier is used by other SCL image functions.
Type: Numeric

list-id
contains the identifier for an SCL list that contains image operation (IMGOP) and image control (IMGCTRL) commands to pass to the function. The commands are processed as the task starts. For command descriptions, see “Commands Used with IMGCTRL” on page 470 and Appendix 1, “Commands Used with the IMGCTRL, IMGOP and PICFILL Functions,” on page 735.
A value of zero means that no list is passed.
Type: Numeric or List

option
specifies a window option:

'NODISPLAY'
prevents the Image window from being created. This can be useful in SCL programs that read, process, or write images without displaying them.

'TOPWINDOW'
causes the Image window to stay on top of the FRAME or SCL window when it is created.
Type: Character

Details The IMGINIT function initializes the image environment. This usually means opening the Image window, but you must use this function even if you plan to manipulate images without displaying them.

When you initialize the Image window, the list that you pass to IMGINIT can include commands to initialize the window size.

Example

Initialize the Image window size:

```
width=700;
height=520;
xpos=0;
ypos=0;

cmdid=makelist();
allcmdid=makelist();
rc=insertc(cmdid,'WSIZE',-1);
rc=insertn(cmdid,width,-1);
rc=insertn(cmdid,height,-1);
rc=insertn(cmdid,xpos,-1);
rc=insertn(cmdid,ypos,-1);
rc=insertl(allcmdid,cmdid,-1);

taskid=imginit(allcmdid,'TOPWINDOW');
```

Start the image task without displaying the Image window:

```
taskid=imginit(allcmdid,'nodisplay');
```

IMGOP

Performs image operations

Category: Image

Syntax

rc=**IMGOP**(*task-id*,*command*,<*other arguments*>);

rc
contains the return code for the operation:

0	successful
>0	not successful

Type: Numeric

task-id
contains the identifier (returned by a previous IMGINIT function) of the task to be operated upon.
Type: Numeric

command
is the command to execute. Valid commands are listed in “Commands Used with IMGOP” on page 473.
Type: Character

other-arguments
lists arguments that are used for a command. The arguments required for a command may include image file attributes. You can specify image file attributes by enclosing the attributes in quotation marks and separating multiple attributes with blanks. For example:

```
rc=imgop(tid,"WRITE",filename,
            "FORMAT=TIFF
            COMPRESS=G3FAX"):
```

Type: Character or Numeric

Commands Used with IMGOP

For detailed information about these commands, see Appendix 1, “Commands Used with the IMGCTRL, IMGOP and PICFILL Functions,” on page 735 .

CONVERT
converts an image to the specified image type and depth.

COPY
copies an image.

CREATE_IMAGE
creates a new image that is stored in memory.

CROP
crops the selected image.

DESTROY
removes an image from memory and from the display.

DESTROY_ALL
removes all images from memory and from the display.

DITHER
dithers an image to a color map.

DITHER_BW
dithers the selected image to a monochrome black and white image.

EXECLIST
executes a list of commands.

FILTER
applies a filter to an image.

GAMMA
applies a gamma value to the selected image.

GENERATE_CMAP
generates a color map for the selected image.

GET_BARCODE
: returns the value of the specified bar code.

GET_COLORS
: returns the RGB values of the index positions of a color map for the selected image.

GET_PIXEL
: returns the pixel value of a specified position in the selected image.

GRAB_CMAP
: grabs the color map from the selected image.

MAP_COLORS
: maps colors to the closest color in the selected color map.

MIRROR
: mirrors an image.

NEGATE
: changes the image to a negative.

PASTE
: displays an image at a specified location.

PASTE_AUTO
: displays an image automatically.

PRINT
: prints an image.

QUANTIZE
: reduces the number of colors used for an image.

QUERYC, QUERYL, and QUERYN
: query information about images.

READ
: reads an image from an external file, a SAS catalog, or a device.

READ_CLIPBOARD
: reads an image from the host clipboard.

READ_PASTE
: reads and displays an image.

READ_PASTE_AUTO
: reads and automatically displays an image.

ROTATE
: rotates an image clockwise by 90, 180, or 270 degrees.

SCALE
: scales an image.

SELECT
: selects the image identifier to be used in other commands.

SET_COLORS
: assigns the RGB values for the index positions of a color map for the current image.

SET_PIXEL
: assigns the pixel value in an image at the specified position.

STANDARD_CMAP
: selects a color map.

THRESHOLD
: converts color images to black and white, using the value that is specified with the THRESHOLD command.

TILE
: replicates the current image into a new image.

UNPASTE
: removes an image from the display.

WRITE
: writes an image to a file or to a SAS catalog.

WRITE_CLIPBOARD
: writes an image to the host clipboard.

IMGTERM

Terminates an image task

Category: Image

Syntax

rc=**IMGTERM**(*task-id*);

rc

contains the return code for the operation:

0	successful
>0	not successful

Type: Numeric

task-id

contains the identifier of the task, which was returned by a previous IMGINIT function.

Type: Numeric

Example

This example shows IMGTERM used within the TERM section:

```
term:
   if (task-id ne 0) then
      rc=imgterm(task-id);
return;
```

IMPORT

Creates a SAS table from an external file

Category: SAS Table

Syntax

name=**IMPORT**(*table-name*,*file*<,'DEFINE'>);

name
contains the name of the last SAS table that was created.
Type: Character

table-name
is the new SAS table to create. If the table already exists, a warning message is displayed when the IMPORT window opens.
Type: Character

file
is the fileref or physical filename of the external file from which data are to be imported. A physical filename must be enclosed in quotation marks. (See the example.)
Type: Character

'DEFINE'
specifies to open the DEFINE window before opening the IMPORT window.
Type: Character

Details

CAUTION:
Blank lines in files can cause problems. Under some host operating systems, blank lines in an external file may adversely affect the ability of the IMPORT function to extract data from the file. △

The IMPORT function returns the name of the last SAS table that it created from an external data file. This function enables users to easily import raw data from an external file into a SAS table.

Two auxiliary windows are associated with IMPORT: the IMPORT window and the DEFINE window.

The IMPORT Window The IMPORT window, which defines the columns for the SAS table, is the primary window for the IMPORT function. The first two lines of the external file are displayed below a ruler in order to help the application user identify the variables for the columns.

If numbers for start and end columns are not supplied, then list input is used, and items in the file must be separated by at least one blank. Users can use the following fields in the IMPORT window to specify information about the table columns:

Name
specifies the name for the table column. This can be any valid SAS name that is up to 32 characters long.

Start/End Column
specify the starting and ending columns of the value in the external file. These fields are optional. You can specify a starting column without specifying an ending column but not vice versa. If you omit the ending column, it is calculated to be one less than the next starting column.

Type
: specifies the data type of the table column. Choose the type by pressing ENTER or clicking with the mouse on the appropriate type.

Format
: specifies a SAS format. Enter a '?' to display a list of some common formats. This field is optional.

Informat
: specifies a SAS informat. Enter a '?' to display a list of some common informats. This field is optional.

Label
: specifies a label for the table column. The label can be up to eight characters long. This field is optional.

IMPORT Window Command Menu The File menu enables you to perform save operations, to import data with the Import Wizard, to export data using the Export Wizard, to perform print operations, to send mail, and to close the window.

The Edit menu enables you to copy marked text, to select or deselect marked text, to set the horizontal and vertical scroll amounts, to perform find operations, and to set the keyfield (the field to search with the next Find command).

The View menu enables you to scroll right and left, to specify the sort order for fields in the window, to display the first record in hexadecimal representation, and to open other SAS windows.

The Tools menu enables you to specify conditions for querying values in the file, to open a Viewtable window, and to run a session of graphics editors, the Report Editor or Text Editor. You can also set a variety of options.

The Data menu enables you to create a SAS table from contents of the external file, to test read the file, to define fields with the Define window, to specify a different external file for input, to specify a new name for the SAS table to create, to display the length of the longest record that has been read, to view the external file, and to edit or browse the data set that was created from the external file.

The DEFINE Window The DEFINE window is displayed when you use the DEFINE option with the IMPORT function or when you select

Data ► Define fields

from the IMPORT window. It displays the first line from the external file, along with a ruler and delimiter lines. On the delimiter lines, you can use < and > to mark the beginning and end of a column, respectively. If the column is only one character wide, use a vertical bar (|).

In order to use the DEFINE window, you must align the data values in columns in the data records.

When you issue the END command to exit from this window, the fields are given default column names and types (numeric or character). The IMPORT window then opens so that you can change the column names and optionally add formats and informats.

Examples

□ Create a new SAS table named MYLIB.NEW using data in the file to which the fileref EXTERN has previously been assigned:

```
name=import('mylib.new','extern');
```

□ Create a new SAS table named MYLIB.NEW using data in the file with the physical name SAMPLE1.DATA. The name of the file is enclosed in quotation marks to indicate that a physical filename, not a fileref, is being passed to the IMPORT function.

```
name=import('mylib.new',"'sample1.data'");
```

Note: The form of the physical filename depends on the host operating system. △

IMPORT

Defines a search path for references to CLASS entries

Category: Catalog, Object Oriented

Syntax

IMPORT *class-specification*;

class-specification
is the two- to four-level name of a CLASS entry. If it is a four-level name, then that CLASS entry is imported. If it is a two-level name, then it is used as the *libname.catalog* prefix for any one- or two- level CLASS entry names in the program. This prefix is concatenated to the one- or two-level CLASS entry name, and that is the CLASS entry that is searched for. The first CLASS entry that is found is the one that is imported. Any subsequent entries that are found are flagged as warnings.

Details The IMPORT statement defines a search path for CLASS entry references in an SCL program so that you can refer to CLASS entries with one- or two-level names instead of having to use a four-level name in each reference. The current catalog is always the first catalog to be searched for a one- or two-level class name, regardless of whether there are any imported catalogs. References to CLASS entries are resolved by the SCL compiler at compile time. The SEARCH function can be used to define a search path that will resolve at run time.

Examples

Example 1: Defining a Search Path Define a MYLIB.MYCAT as a search path for the program.

```
   /* All the program's classes  */
   /* are defined in MYLIB.MYCAT *
IMPORT mylib.mycat;
   /* collobj1 is defined in       */
   /* mylib.mycat.collection.class */
DECLARE Collection c1=_new_ Collection();
   /* collobj2 is defined in              */
   /* mylib.mycat.OrderedCollection.class */
DECLARE OrderedCollection c2=_new_ OrderedCollection();
```

Example 2: Importing a Class from Another Catalog This example imports class X from another catalog. Class Y is in the current catalog.

```
X.SCL
class work.cat.x;
  public num n;
endclass;

Y.SCL
import work.cat2;
class y;
  m: method o: x;
    o.n=99;
  endmethod;
endclass;

Z.SCL
init:
  import work.cat2;
  dcl y yobj=_new_ y();
  dcl x xobj=_new_ x();
  yobj.m(xobj);
  put xobj.n=;
return;
```

This example should produce the following output:

```
xobj.n=99
```

See Also

"DECLARE" on page 330

"SEARCH" on page 635

INFORMAT

Verifies that the specified informat is valid

Category: Formatting

Syntax

rc=**INFORMAT**(*informat*,*type*);

rc

contains the return code for the operation:

1	successful
0	not successful

Type: Numeric

informat

is an informat that is supplied by SAS or created using the FORMAT procedure. The name of an informat must include a period (for example, **date7.** or **comma10.2**).

Type: Character

type
specifies the type of the informat:

'C'	character
'N'	numeric

Type: Character

Details If the specified informat is not known to the SAS session, then the operation is unsuccessful. The function verifies that valid widths are specified for informats.
See *SAS Language Reference: Dictionary* for details about informats.

Examples

Example 1: Verifying the Validity of a Character Informat Verify that **$MYFMT.** is a valid character informat that has been defined for the current SAS session. (The value returned to the variable RC is 1.)

```
rc=informat('$myfmt.','c');
```

Example 2: Verifying the Validity of a Numeric Informat Verify that **5.6** is not a valid informat for numeric values. (The value returned to the variable RC is 0.)

```
rc=informat('5.6','n');
```

See Also

"FORMAT" on page 424

INITROW

Initializes the Table Data Vector (TDV) for a SAS table to missing values

Category: SAS Table

Syntax

rc=**INITROW**(*table-id*);

rc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

table-id
is the identifier that was assigned when the SAS table was opened.
Type: Numeric

Details INITROW initializes the TDV to missing values. This prevents bad data from being written to a table row when you do not explicitly assign values to columns with the PUTVARC or PUTVARN function and you use the APPEND function with the NOINIT option.

Example

Open the table whose name is saved in the variable TABLE, initialize a new row to missing, and then write a value for the NAME column. When the new row is appended to the data table, the column NAME is initialized to the value JOHN while values in all other columns are set to missing.

```
tableid=open(table);
rc=initrow(tableid);
call putvarc(tableid,varnum(tableid,'name'),'JOHN');
rc=append(tableid,'noinit');
rc=close(tableid);
```

INPUTC and INPUTN

Read a character value using an informat

Category: Formatting

Syntax

char-value=**INPUTC**(*char-string*,*char-informat*);

num-value=**INPUTN**(*char-string*,*num-informat*);

char-value
contains *char-string* with the character informat applied to it.
Type: Character

num-value
contains *char-string* converted to a numeric value with the numeric informat applied to it.
Type: Numeric

char-string
is the character string to be read.
Type: Character

char-informat
is the character informat to use for reading *char-string*.
Type: Character

num-informat
is the numeric informat to use for reading *char-string*.
Type: Numeric

Details INPUTC and INPUTN both read a character value. However, INPUTC applies a character informat and returns a character value, and INPUTN applies a numeric informat and returns a numeric value. These functions are similar to the INPUT function in the DATA step.

Note: Dot notation cannot be used with the INPUTC or INPUTN functions. This restriction is necessary to allow proper parsing of the *char-informat* and *num-informat* parameters. △

For more information about using an informat to read a value, see the INPUT function for the DATA step in *SAS Language Reference: Dictionary*.

Examples

Example 1: Using the INPUTC Function Read the character variable NAME, using the $UPCASE3. informat:

```
name='sas';
cval=inputc(name,'$upcase3.');
put cval=;
```

This program produces the following output:

```
cval=SAS
```

Example 2: Using the INPUTN Function Read the character variable AMOUNT, containing the value $20,000.00, into the numeric variable SALARY, using the COMMA10.2 informat:

```
amount='$20,000.00';
informat='comma10.2';
salary=inputn(amount,informat);
put salary=;
```

This program produces the following output:

```
salary=20000
```

Example 3: Using INPUTN with a DATE Value Read the value in DATE and apply the JULIAN8. informat:

```
date='90091';
ndate=inputn(date,'julian8.');
put ndate=;
```

This program produces the following output:

```
ndate=11048
```

See Also

"PUTC and PUTN" on page 605

INSERTC, INSERTL, INSERTN, and INSERTO

Insert a value into an SCL list

Category: List

Syntax

rc=**INSERTC**(*list-id*,*cval*<,*index*<,*name*>>);

rc=**INSERTL**(*list-id*,*sublist-id*<,*index*<*name*>>);

rc=**INSERTN**(*list-id*,*nval*<,*index*<*name*>>);

rc=**INSERTO**(*list-id*,*object-id*<,*index*<*name*>>);

rc
is the *list-id*, which is the identifier of the modified list.
Type: Numeric

list-id
is the identifier of the list into which to insert the item. An invalid *list-id* produces an error condition.
Type: Numeric or List

cval
is the character value to insert into the list with INSERTC.
Type: Character

sublist-id
is the identifier of the sublist to insert into the list with INSERTL. An invalid *sublist-id* produces an error condition.
Type: Numeric

nval
is the numeric value to insert into the list with INSERTN.
Type: Numeric

object-id
is the identifier of the object to insert into the list with INSERTO. An invalid *object-id* produces an error condition.
Type: Numeric or Object

index
is the position at which to insert the item into the list. The position can be specified as a positive or negative number. By default, *index* is 1 (the first item). If *index* is a positive number, then the item is at position *index* from the beginning of the list. If *index* is a negative number, then the item is at position ABS(*index*) from the end of the list. *Index* must be in the range [–(*n*+1),–1] or [1,*n*+1] where *n* is the length of the list. An error condition results if the absolute value for *index* is zero or if it is greater than the number of items in the list.
Type: Numeric

name
is the name to assign to the item. If *name* is omitted, a name is not assigned to the item.
Type: Character

Details The item is inserted such that after you insert an item at position *index*, you can retrieve it from position *index* with any of these functions.

These functions do not make a copy of the list. The insertion is performed in place. You can append an item to an SCL list of length *n* by inserting at *index*=n+1 or at *index*=–1.

Note: The return value of these functions is not used to indicate whether an error has been detected. When an error occurs, the program simply halts. △

An error condition results if

- the list has any of the following attributes:
  - NOUPDATE
  - FIXEDLENGTH

    - CHARONLY, and you use an insert function other than INSERTC
    - NUMONLY, and you use an insert function other than INSERTN
    - SASNAMES, and *name* is omitted or is not a valid SAS name
    - NODUPNAMES, and *name* duplicates the name of a list item.
- the absolute value for *index* is greater than 1 plus the number of items in the list or is 0.
- you attempt to insert a local list into a global list with INSERTL.

You can use HASATTR to check the attributes of a list or list item. To change attributes, use SETLATTR.

Examples

Example 1: Using the INSERTC Function Insert **CANADA** as the third item in the list:

```
listid=insertc(listid,'CANADA',3);
```

After this insertion, return the value that was third in the list before the insertion of **CANADA** shifted the value from the third to the fourth position:

```
cval=getitemc(listid,4);
```

Example 2: Using the INSERTL Function Insert the sublist NEWLIST as the third item from the end of the list:

```
listid=insertl(listid,newlist,-3);
```

Example 3: Using the INSERTN Function Assume that the list MYLIST contains four items, named **A**, **B**, **C**, and **D**, with the values 1, 4, 9, and 16, respectively. Insert two new items: a string at the default position 1 (the beginning of the list), and a number at position –1 (the end of the list). The new number is given the name **E**.

```
call putlist(mylist,'Before: ',0);
mylist=insertc(mylist,'Squares');
mylist=insertn(mylist,25, -1,'E');
call putlist(mylist,'After: ',0);
```

This program produces the following output:

```
Before: (A=1
         B=4
         C=9
         D=16
         )[3]
After: ('Squares'
        A=1
        B=4
        C=9
        D=16
        E=25
        )[3]
```

Note: [3] is the list identifier that was assigned when this example was run and may be different each time the example is run. △

Example 4: Using the INSERTO Function Create the list MYLIST, insert the item whose identifier is stored in the variable MYOBJECT, and assign the name **My Object** to the item:

```
declare sashelp.fsp.object myobject, =_new_ sashelp.fsp.object(),
        list mylist;
mylist=makelist();
rc=inserto(mylist,myobj,-1,'My Object');
```

See Also

"GETITEMC, GETITEML, GETITEMN, and GETITEMO" on page 442

"GETNITEMC, GETNITEML, GETNITEMN, and GETNITEMO" on page 445

"POPC, POPL, POPN, and POPO" on page 593

"SETITEMC, SETITEML, SETITEMN, and SETITEMO" on page 651

"SETNITEMC, SETNITEML, SETNITEMN, and SETNITEMO" on page 661

INSTANCE

Creates an object and returns its identifier

Category: Object Oriented

Syntax

object-id=**INSTANCE**(*class-id*<,*arg*>);

object-id
contains the identifier that was assigned to the new object.
Type: Numeric or Object

class-id
is the identifier for the class, which is returned by the LOADCLASS function.
Type: Numeric

arg
is an argument to pass to the _init method.
Type: Numeric

Details When creating an object (or instance) of a class, INSTANCE sends the _init method for the specified class to the new instance and passes *arg* as an argument to the _init method. If *arg* is not specified, then no argument is passed to the _init method. To indicate that the numeric parameter is optional, the _init method of all classes should use the OPTIONAL= option in the METHOD statement or ENTRY statement.

A common practice is to use an SCL list as *arg*. You can then pass an arbitrary list of data which will be accessible in the _init method.

To delete an object that was created with INSTANCE, use its _term method. For example:

```
dcl object objectid;
objectid=instance(classid);
objectid._term();
```

You cannot use the INSTANCE function to create instances of the Frame class.

For more information about classes and methods, see the documentation for SAS/AF classes.

Example

Load a class named Queue, a subclass of the Object class, and create two instances of the Queue class. The Inqueue class is created with a maximum number of items. The Outqueue class does not have a maximum.

```
queue=loadclass('applib.classes.queue');
inqueue=instance(queue, max_items);
outqueue=instance(queue);
```

Assume that the _init method of the Queue class is declared as

```
_init: method optional= maxItems 8;
   ...more SCL statements...
endmethod;
```

See Also

"APPLY" on page 252

"ENTRY" on page 369

"LOADCLASS" on page 525

"LOADRES" on page 526

"METHOD" on page 540

"_NEO_" on page 559

"NOTIFY" on page 572

"SEND" on page 644

"SUPAPPLY" on page 679

"SUPER" on page 681

INTERFACE

Defines a group of abstract methods shared by the related classes

Category: Object Oriented

Syntax

INTERFACE *interface-name*
  <EXTENDS *interface-name*>
  </ (*interface-optional-clause*)>;
  <(*limited-method-declaration-statements*)>

ENDINTERFACE;

interface-name
is the name of the interface that you are defining, which can be specified as a one- to four-level name. The entry type is INTERFACE.

EXTENDS-*interface-name*
specifies the parent interfaces as a one- to four-level name. An interface can inherit from parent interfaces. If you do not use the EXTENDS clause, the *parent-interface* defaults to SAS HELP.FSP.INTERFACE.CLASS.

interface-optional-clause
specifies options for the interface. You must put the list inside the parentheses that follow a / (slash). Separate multiple options with commas. The only option currently supported is

Description = *description*
is a description of the INTERFACE entry.

limited-method-declaration-statements
defines the interface methods. Declare the methods as follows:

```
method-label-name: METHOD <argument-list><OPTIONAL=argument-list>
</(method-options)>;
```

method-label-name
can be up to 32 characters and has the same restrictions as an SCL label.

argument-list
list one or more sets of arguments, with each set specified as follows:

```
var-list <:INPUT|UPDATE|OUTPUT>:data-type(length)
```

var-list
lists one or more variables to contain values that are passed in from a method call using either dot notation or the METHOD, SEND, SUPER, APPLY or SUPAPPLY routine or function. Variables can also be reference arrays. Reference array dimensions are specified by '*'. Use commas to separate '*' for multiple dimensions. The actual size of the reference array will be determined at run time based on the dimensions specified in the array parameter of the calling method. For more information, see "ARRAY" on page 254 and Example 3 on page 546.

INPUT | I
specifies that, at run time, the variable contains the value that is copied from the corresponding parameter of the calling method. However, when the program finishes, the value is not copied back to the calling method.

UPDATE | U
specifies that, at run time, the variable contains the value that is copied from the corresponding parameter of the calling method. When the program finishes, the value is copied back to that parameter.

OUTPUT | O
specifies that, when the program finishes, the value is copied back to the corresponding parameter in the calling program. An error condition results if the corresponding parameter in the calling program is a constant, because a constant cannot receive a value.

data-type
specifies the type of data that the variable will contain. A named data type (for example, CHAR or LIST) must be preceded by the **:** delimiter. The delimiter is optional for unnamed data types (for example, $).

CHAR<**(***length***)**>
specifies that the variable will contain character data. *Length* can be 1 to 32,767 characters. If *length* is not provided, the default length is 200.

LIST
specifies that the variable will contain an SCL list identifier.

NUM
specifies that the variable will contain a numeric value.

OBJECT
specifies that the variable will contain the identifier for an object when it is defined at run time.

This type causes the SCL compiler to generate extra conversion instructions. Consequently, you should use it only when necessary so as to achieve optimal run-time performance.

class-name
specifies that the variable will contain the identifier for an object of the class specified in *class-name*. *Class-name* must be a three- or four-level name unless an IMPORT statement has specified the libref and catalog. In that case, the name can be a one- to four-level name. If the entry type is not specified, it is assumed to be CLASS.

interface-name
specifies that the variable will contain the identifier for an object of the class that supports the interface specified in *interface-name*. *Interface-name* must be a three– or four-level name unless an IMPORT statement has been used to specify the libref and catalog. In that case, the name can be a one- to four-level name.

If the entry type is not specified and a class with that name does not exist, the default entry type of INTERFACE is assumed.

To be compatible with the applications built in earlier releases of SAS software, the **:** delimiter is optional for variables that have been declared with unnamed data types (for example, $), but it is required for variables that have been assigned named data types. The following example shows a variety of data type declarations, including reference arrays that use * as the dimensions:

```
mymethod: method
 char1  : Char(20)
   char2  : Char(10)
   char3  :input :char(50)
   charArr(*):u:char /* a reference array */
   num1   : num
   num2   : num
   num3   : num
   numArr(*):num     /* a reference array */
   myList :list
   myObj  :object
   myCol :Sashelp.Fsp.Collection.class ;
```

Type: Character

length
is a numeric constant that specifies the length of the preceding variable or variables. The length of a character variable does not have to match the length of the corresponding passed parameter. SCL pads or truncates as necessary. When a

length is specified for a variable that is declared as CHAR, the length specification must be enclosed in parentheses.
Type: Character

OPTIONAL=
enables you to specify one or more optional arguments that are used only if the calling program supplies the corresponding parameters in the parameter list of the calling routine. If corresponding parameters are not supplied, then the optional arguments are initialized to missing values.

method-options
specify options for an interface method. You must put the list inside parentheses that follow a / (slash). The only option currently supported is Description = *description*, which may be used to provide a description of the method.

Details In order to group related classes which share similar method names, a superclass can be created. One approach to create the superclass is to use the INTERFACE block. You can use the INTERFACE and ENDINTERFACE statements to create an INTERFACE block that contains method definitions. Method implementations are not allowed in the INTERFACE block, so all methods are defined as abstract methods.

Interfaces describe “contracts” in a pure, abstract form, but an interface is interesting only if a class supports it. Any class that supports an interface must implement all of the methods defined in the interface. Since the INTERFACE block does not contain any method implementations, class developers can use the INTERFACE block to reflect the high- level design of their applications. Any class can also be specified with the “required” interface using the “Required-Clause”. The SCL compiler will generate information that will be used to validate whether the actual method used at run time matches the required interface.

To create an interface from an SCL entry that contains an INTERFACE block, you must issue either the SAVECLASS command or from FILE menu Save Class Pmenu. This compiles the entry and creates the INTERFACE entry that is specified by *interface-name*. This is equivalent to using the Interface Editor to interactively create an INTERFACE entry. However, the Interface Editor provides a tabular view of the interface, whereas the INTERFACE statement in SCL provides a language view of the interface. For maintenance purposes, the existing INTERFACE entry can also be converted to SCL syntax by using the CreateSCL function. For more detailed information, please refer to the CREATESCL“CREATESCL” on page 316 function.

Examples

Example 1: Defining a Simple Interface The following INTERFACE block defines a simple myWidget interface with four GUI methods.

```
interface work.a.myWidget.interface;
     refresh: method;
     needRefresh: method;
     select: method;
     tab: method n1:NUM n2:NUM;
endinterface;
```

The following INTERFACE block defines an I/O interface with two I/O methods.

```
interface work.a.myIO.
interface;
  read: method string buffer;
  write: method string buffer;
```

```
endinterface;
```

Example 2: Defining Classes That Support Interfaces The INTERFACE blocks cannot contain method implementations, so any class that supports an interface must implement all of the methods defined in the interface. The following class implements methods that are defined in **`work.a.myIO.interface`**:

```
class work.a.model.class supports work.a.myIO.interface;
  read: method string buffer
         / (scl='work.a.model.scl');
  write: method string buffer
          / (scl='work.a.model.scl');
  /* The following method is not in the interface */
  myMethod: method n:Num
          / (scl='work.a.myscl.scl');
endclass;
```

The following class supports both the myWidget interface and the myIO interface.

```
class work.a.modelViewer.class
      supports work.a.myWidget.interface,
                work.a.myIO.interface;
  refresh: method / (scl='work.a.mv.scl');
  needRefresh: method / (scl='work.a.mv.scl');
  select: method / (scl='work.a.mv.scl');
  tab: method n1:NUM n2:NUM
        / (scl='work.a.mv.scl');
  read: method string buffer
         / (scl='work.a.model.scl');
  write: method string buffer
          / (scl='work.a.model.scl');
  myMethod: method n: num
          / (scl='work.a.myscl.scl');
endclass;
```

Example 3: Defining Classes That Require Interfaces The following class requires both the myWidget interface and the myIO interface. By specifying which interfaces are required, you allow the SCL compiler to generate information that will be used to validate whether the actual methods used at run time matched the required interface.

```
class work.a.myRequired.class
      Requires work.a.myWidget.interface,
               work.a.myIO.interface;
   ...Other implementations...
endClass;
```

See Also

"CLASS" on page 277

IOPTION

Returns options for index columns and key columns

Category: SAS Table

Syntax

options=**IOPTION**(*table-id*,*key-name*);

options
contains options for the specified index *key-name,* separated by a blank:

MISSING
The index can contain missing values.

NOMISS
The index does not contain missing values.

NONUNIQUE
The index can contain non-unique values.

UNIQUE
The index contains only unique values.
Type: Character

table-id
is the identifier that was assigned when the table was opened. If *table-id* is invalid, the program halts.
Type: Numeric

key-name
is the name of an index key.
Type: Character

Details An *index* is an auxiliary data structure used to speed up the selection of records that are specified by the value of a column.

You can create indexes using

- the ICREATE function in SCL
- the DATASETS and SQL procedures in base SAS software
- SAS/IML software
- the CONTENTS function in SCL
- the ACCESS procedure in SAS/ACCESS software.

When an error occurs, IOPTION returns a blank string.

Example

Return the options of the defined key index ITEMTYPE for the SAS table WORK.DATAONE. If the value returned to the OPTIONS column is blank, then the message returned by the SYSMSG function is displayed on the message line.

```
tableid=open('work.invoice','i');
options=ioption(tableid,'itemtype');
if (options=' ') then _msg_=sysmsg();
```

See Also

"CONTENTS" on page 299

"ICREATE" on page 464

"IDELETE" on page 468

"ISINDEX" on page 494

"IVARLIST" on page 498

ISACTIVE

Returns the number of the active button in a radio box or check box or the active station in a choice group

Category: Control or Field

Syntax

station=**ISACTIVE**(*var-name*<,*row*>);

station
contains the status of a selection:

>0	the number of the button or station that is active
0	no button or station is active

Type: Numeric

var-name
is the radio or choice group to be tested.
Type: Character

row
is the row number when the button or choice group is in the scrollable section of an extended table in a PROGRAM entry. Do not specify *row* in programs for FRAME entries. Specify *row* only when you want to check the active station from outside the extended table's getrow or putrow section.
Type: Numeric

Details You cannot use ISACTIVE in FSEDIT or FSVIEW programs.
Window controls can also use the _isActive method.

Example

Suppose your application has a radio box named HOBBY in which the third button displays the value **TENNIS**. Branch to an appropriate program when a user selects the **TENNIS** station (either by pressing ENTER or by clicking the mouse button).

```
if (isactive('hobby')=3)
then call display('tennis.frame');
```

See Also

"ACTIVATE" on page 249

"GRAY" on page 457

"ISGRAY" on page 493

"UNGRAY" on page 693

ISGRAY

Reports whether a FRAME entry control or choice group is grayed

Category: Control or Field

Syntax

rc=**ISGRAY**(*var-name*<,*station*<,*row*>>);

rc
contains the return code for the operation:

1	the specified station is grayed
0	the specified station is not grayed, or no station in the choice group is grayed
m	the number of stations that are grayed, when an entire choice group is not grayed
–*n*	the total number of stations, when an entire choice group is grayed.

Type: Numeric

var-name
is the window control or choice group to be tested.
Type: Character

station
is the number of the choice group station or the button in a radio box. *Station* must be greater than 0 and must be no greater than the total number of buttons in a radio box or the number of stations in a choice group.
Type: Numeric

row
is the number of the row when the control or choice group is in the scrollable section of an extended table. *Row* is valid for PROGRAM entries. Specify *row* only when you want to determine whether the station is gray from outside the extended table's getrow or putrow section.
Type: Numeric

Details Window controls can also use the _isGray method.

Because choice groups can be defined only in SAS/AF software, you cannot use ISGRAY in FSEDIT or FSVIEW programs.

Examples

Example 1: Determining Whether an Extended Table Row Is Grayed Test whether the choice group WINE at the third row of an extended table is grayed:

```
if (isgray('wine',0,3)) then
      do;
       ...SCL program statements...
      end;
```

Example 2: Finding the Number of Stations in a Choice Group Find out how many stations are defined for a choice group:

```
rc=gray('wine');
   total=abs(isgray('wine'));
```

See Also

"ACTIVATE" on page 249

"GRAY" on page 457

"ISACTIVE" on page 492

"UNGRAY" on page 693

ISINDEX

Returns the type of index for a SAS table column

Category: SAS Table

Syntax

index=**ISINDEX**(*table-id*,*col-name*);

index
contains the type of the index:

(blank)
No index has been created for the specified column.

BOTH
The column is a member of both simple and composite indexes.

COMP
The column is a member of a composite index.

REG
The column is a regular (simple) index.
Type: Character

table-id
is the identifier that was assigned when the table was opened. If *table-id* is invalid, the program halts.
Type: Numeric

col-name
is the name of a column in the SAS table. If *col-name* is not a column in the table, then ISINDEX returns a blank value.
Type: Character

Details An *index* is an auxiliary data structure used to assist in the location (that is, selection) of rows that are specified by the value of a column. An index is called a *simple index* if it contains the value of only one column. A *composite index* merges the values for more than one column to form a single value. A given SAS table can have multiple simple indexes, composite indexes, or any combination of these.

You can create indexes using

- the ICREATE function in SCL
- the DATASETS and SQL procedures in base SAS software
- SAS/IML software
- the CONTENTS function in SCL
- the ACCESS procedure of SAS/ACCESS software.

Example

Return the type of index for the FNAME column in the SAS table WORK.DATAONE:

```
dsid=open('work.dataone','i');
ixtype=isindex(dsid,'fname');
```

See Also

"CONTENTS" on page 299

"ICREATE" on page 464

"IDELETE" on page 468

"IOPTION" on page 490

"IVARLIST" on page 498

ISSEL

Returns the selection number for a specified row of a selection list

Category: Extended Table

Syntax

selection=**ISSEL**(*row*);

selection
contains the row's selection number, or 0 if the row is not selected.
Type: Numeric

row
is the number of the row that is being queried.
Type: Numeric

Details You can use the ISSEL function in two ways:

- □ to determine the order in which a certain row was selected
- □ to determine whether a row is being selected or deselected.

Because you can define extended tables only with SAS/AF software, you cannot use ISSEL in FSEDIT or FSVIEW programs. ISSEL is valid only for PROGRAM entries. Window controls must use the _issel method.

In order for an extended table to be considered a selection list, you must specify a number of selections in the SETROW routine.

Example

Suppose that your application has a selection list with ten rows and that the user has just selected row 3 and then row 5. If ISSEL is called with the *row* argument equal to 5, then the value 2 is returned for *selection*, because row 5 was the second selection.

You can also use ISSEL in the putrow section of an extended table application to test whether a row is selected. Call the function for the desired row and check the *selection* value to see whether its value is positive (the row has been selected) or zero (the row has been deselected):

```
PUTROW:
   if (issel(_CURROW_)) then
      do;
        ...SCL statements to process the selected row...
      end;
   else
      do;
        ...SCL statements to process the deselected row...
      end;
return;
```

See Also

"NSELECT" on page 573

"SELECT" on page 640

"SELECTED" on page 642

"UNSELECT" on page 696

ITEM

Specifies the classes on the server that can be accessed by applications on the client

Category: Object Oriented

Syntax

ITEM *class-name*;

class-name
is the name of a class that you want included in the class package that you are defining. The class name is a one- to four-level name of the form *library.catalog.member*.CLASS.

Details

ITEM statements define the classes on the server whose methods and attributes can be accessed by applications on the client.

See Also

"PACKAGE" on page 585

ITEMTYPE

Reports the type of an item in an SCL list

Category: List

Syntax

type=**ITEMTYPE**(*list-id*<,*index*>);

type
contains the type of the specified item:

C	The item is a character item.
N	The item is a numeric item.
L	The item is a sublist item.
O	The item is a component item.

Type: Character

list-id
is the identifier of the list containing the item whose type is returned by ITEMTYPE. An invalid *list-id* produces an error condition.
Type: Numeric or List

index
is the position of the item in the list. The position can be specified as a positive or negative number. By default, *index* is 1 (the first item). If *index* is a positive number, then the item is at position *index* from the beginning of the list. If *index* is a negative number, then the item is at position ABS(*index*) from the end of the list. An error condition results if the absolute value for *index* is zero or if it is greater than the number of items in the list.
Type: Numeric

Details An item's type depends on which function was used to create the item:

C (character) items
are created by SETITEMC, SETNITEMC, INSERTC.

N (numeric) items
: are created by SETITEMN, SETNITEMN, INSERTN, MAKELIST, MAKENLIST.

L (sublist) items
: are created by SETITEML, SETNITEML, INSERTL.

O (component) items
: are created by SETITEMO, SETNITEMO, INSERTO.

See Also

"INSERTC, INSERTL, INSERTN, and INSERTO" on page 482

"MAKELIST" on page 535

"MAKENLIST" on page 536

"SETITEMC, SETITEML, SETITEMN, and SETITEMO" on page 651

"SETNITEMC, SETNITEML, SETNITEMN, and SETNITEMO" on page 661

IVARLIST

Returns the column names for an index key

Category: SAS Table

Syntax

varlist=**IVARLIST**(*table-id*,*key-name*);

varlist
contains one or more index columns (separated by a blank) for the specified key, or a blank if *key-name* is invalid.
Type: Character

table-id
contains the identifier that was assigned when the table was opened. If *table-id* is invalid, the program halts.
Type: Numeric

key-name
is the name of an index key.
Type: Character

Details An *index* is an auxiliary data structure used to speed up the selection of records that are specified by the value of a column.

An index is called a *simple index* if it contains the value of only one column. A *composite index* merges the values for more than one column to form a single value. A given SAS table can have multiple simple indexes, composite indexes, or a combination of these.

You can create indexes using

- the ICREATE function in SCL
- the DATASETS and SQL procedures in base SAS software
- SAS/IML software

- □ the CONTENTS function in SCL
- □ the ACCESS procedure of SAS/ACCESS software.

Note: Because some engines now support mixed-case filenames, IVARLIST now retains the cases of the returned selected items. If your application contains code that assumes that the returned selection is in uppercase, your application may fail. You may need to modify your application. For example, you can use the UPCASE* function to convert the returned selection to uppercase:

```
if (upcase(ivarlist(dsid, 'TESTNDX'))='NDXVAR'
```

If the application cannot be modified, you may need to specify the VALIDVARNAME=V6 system option when you run the application to ensure that the selections returned from the IVARLIST function are in uppercase. △

Example

Return the column list that is indexed for the key ITEMTYPE in the SAS table MYLIB.DATAONE. Assume that ITEMTYPE is a simple index (that is, it contains the values of only one column). The returned VARLIST contains the string ITEMTYPE.

```
tableid=open('mylib.dataone','i');
varlist=ivarlist(tableid,'itemtype');
```

See Also

VALIDVARNAME system option in *SAS Language Reference: Dictionary*

"ICREATE" on page 464

"IDELETE" on page 468

"IOPTION" on page 490

"ISINDEX" on page 494

KEYCOUNT

Returns the number of rows that meet the criteria specified by an index key

Category: SAS Table

Syntax

nrow=**KEYCOUNT**(*table-id*);

nrow
contains the number of rows that meet the criteria, or <0 if an error occurred. The error message can be retrieved by using SYSMSG.
Type: Numeric

* For documentation for the UPCASE function, refer to *SAS Language Reference: Dictionary*.

table-id
is the identifier that was assigned when the table was opened. If *table-id* is invalid, the program halts.
Type: Numeric

Details KEYCOUNT returns the number of rows that meet the criteria specified by the index key column. The index key column was specified with the last SETKEY function that was used on the table. After KEYCOUNT executes, the table points to the first row that meets the criteria defined by the last SETKEY function. Use FETCH or FETCHOBS to read the row.

CAUTION:
Using KEYCOUNT with composite keys may show a larger number of rows matching the search criteria than you expect. Using a composite key with SETKEY operates the same way as the WHERE function only when the condition is EQ. The value returned when the condition is EQ is the same as if the columns specified in the composite key are connected by WHERE conditions using AND or ALSO. (See Example 1.)

For all other conditions (GT, GE, LT, or LE) specified with SETKEY for a composite key, the composite key columns are concatenated to form the index key. The number that the KEYCOUNT function returns is the number of rows in the table that satisfy the composite key. For example, if the composite index consists of columns GENDER and AGE and the condition is GT (greater than), the values to search for are concatenated such that key values of **`F`** for GENDER and **`13`** for AGE yield an index key of **`F13`**. Because the search is performed on the concatenated values, some values that you did not expect may meet the search condition . For example, key values of **`M`** for GENDER and **`11`** for AGE meet the search condition, because the string **`M11`** is considered greater than the string **`F13`**. (See Example 2.) △

Examples

Suppose you have the following examples which use rows from the WORK.CLASS table. Create a simple index for the table WORK.CLASS, using ICREATE or the DATASETS procedure, with AGE as the index column. Also, create a composite index for WORK.CLASS called COMP that consists of columns GENDER and AGE.

Example 1: Using a Simple Index Key Set up a search criteria of AGE=13. SETKEY specifies that the key column is AGE and the condition is equality.

```
/*  Locate rows where 'age = 13'   */
   tableid = open( 'work.class', 'v' );

      /* Create the simple index */
   rc = icreate(tableid,'age');
   name = ''; gender = '';
   age = 13;
   call set(tableid);
   rc = setkey(tableid,'age','eq');
   nrow = keycount(tableid);

   if (nrow < 0) then _msg_ = sysmsg();
   else
      do;
         put 'Number of rows found:' nrow;
         do while (fetch(tableid) ne -1);
            put name= gender= age=;
         end;
      end;
```

This program produces the following output:

```
Number of rows found: 3
   name=Alice gender=F age=13
   name=Becka gender=F age=13
   name=Jeffery gender=M age=13
```

Example 2: Using a Composite Index Key with Condition 'EQ' Set up search criteria of GENDER=F and AGE=13. SETKEY specifies that the key is named COMP and the condition for the search is equality.

```
/* Locate rows where 'gender="F"' */
/* and 'age=13' */
   tableid = open( 'work.class', 'v' );

      /*  Create index */
   rc = icreate(tableid,'comp','gender age');
   name = ''; gender = 'F'; age = 13;
   call set(tableid);
   rc = setkey(tableid,'comp','eq');
   nrow = keycount(tableid);

   if (nrow < 0) then _msg_ = sysmsg();
   else
   do;
      put 'Number of rows found:' nrow;
      do while (fetch(tableid) ne -1);
         put name= gender= age=;
      end;
   end;
```

This program produces the following output:

```
Number of rows found: 2
   name=Alice gender=F age=13
   name=Becka gender=F age=13
```

Example 3: Using a Composite Index Key with Condition 'GT' Set up search criteria of GENDER=F and AGE greater than 13. SETKEY specifies that the key is named COMP and the condition for the search is greater-than. This example illustrates the unexpected results returned by KEYCOUNT when you use composite index keys and SETKEY using a 'GT' argument.

```
/*  Locate rows where 'genderage' > 'F13'  */
   tableid = open( 'work.class', 'v' );

      /* Create index */
   rc=icreate(tableid,'comp','gender age');
   name = ''; gender = 'F'; age = 13;
   call set(tableid);
   rc = setkey(tableid,'comp','gt');
   nrow = keycount( tableid);

   if (nrow < 0) then _msg_ = sysmsg();
   else
   do;
```

```
        put 'Number of rows found:' nrow;
        do while (fetch(tableid) ne -1);
           put name= gender= age=;
        end;
     end;
```

This program lists 14 rows from the indexed table that met the search criteria of GENDER||AGE>=F13.

```
  Number of rows found: 14
     name=Gail gender=F age=14
     name=Tammy gender=F age=14
     name=Mary gender=F age=15
     name=Sharon gender=F age=15
     name=Thomas gender=M age=11
     name=James gender=M age=12
     name=John gender=M age=12
     name=Robert gender=M age=12
     name=Jeffrey gender=M age=13
     name=Alfred gender=M age=14
     name=Duke gender=M age=14
     name=Guido gender=M age=15
     name=William gender=M age=15
     name=Philip gender=M age=16
```

You can see that James at AGE=12 does not meet the SETKEY requirement of AGE > 13 and GENDER > 'F'. However, his row was selected because the values were concatenated before the comparison was made.

See Also

"SETKEY" on page 653

LASTCMD

Returns the text of the last command that was issued from the application window

Category: Command

Syntax

cmdtext=**LASTCMD**();

cmdtext
contains the text of the last command that was issued from the application window.
Type: Character

Details If the command contains multiple words, only the first word is returned.
LASTCMD is usually used in conjunction with CONTROL ENTER, ALWAYS, or ALLCMDS.

Example

Retrieve the last command that was issued in the window and display a message, based on that command name:

```
INIT:
   control always;
return;
MAIN:
   cmd=lastcmd();
   if cmd='GO' then
      _msg_='Last command was '||cmd;
return;
```

See Also

"WORD" on page 725

LASTKEY

Returns the number of the last function key that was pressed from the application window

Category: Keys

Syntax

keynum=**LASTKEY**();

keynum
contains the number of the function key that was pressed from the application window, or 0 if ENTER was pressed.
Type: Numeric

Details The returned value is the ordinal position of the key definition in the KEYS window. In order for this function to work, you must have a window variable or text in the DISPLAY window.

To retrieve the name of the last function key pressed by a user, use FKEYNAME.

LASTKEY is used in conjunction with CONTROL ENTER, ALWAYS, and ALLCMDS. LASTKEY does not retrieve the number of a function key that has a global command assigned to it.

Example

Return the number of the last function key that a user pressed. (This example requires a window with at least one window variable.)

```
INIT:
   control enter;
return;

MAIN:
   keynum=lastkey();
   if (keynum ne 0) then
      put 'Last function key is ' keynum;
   else
      put 'Last function key is not defined
           or the ENTER key was pressed';
return;
```

See Also

"FKEYNAME" on page 413

"GETFKEY" on page 440

"SETFKEY" on page 649

"CONTROL" on page 302

LEAVE

Stops processing the current DO group or DO loop and resumes with the next statement in sequence

Category: Control Flow

Comparisons: SAS Statement with limitations in SCL

Syntax

LEAVE <*label*>;

label
is the name of a program label that is associated with the DO group.
Type: Character

Details The LEAVE statement is provided in SCL to control the execution of DO groups. When you need to force the statements in a DO group to stop executing, you can use the LEAVE statement to stop executing statements in a DO group and to start executing a statement that is outside of that DO group.

Note: In DATA step code, the LEAVE statement stops processing only the current DO loop. In SCL code, the LEAVE statement stops processing the current DO loop *or DO group*, whichever is closest. For example, suppose your code contains a DO loop that contains DO groups:

```
do n=1 to 5;     /* DO loop */
    if n=3 then do; leave; end;        /* DO group */
  put n=;
end;
```

When this code is compiled and run as part of an SCL program, the output is:

```
n=1
n=2
n=3
n=4
n=5
```

When this code is submitted as part of a DATA step, the output is:

```
n=1
n=2
n=3
```

See “DO” on page 356 for more information on DO groups and DO loops. △

For details about the LEAVE statement in the base SAS language, see *SAS Language Reference: Dictionary*.

Examples

Example 1: LEAVE Statements Without Label Names If a LEAVE statement does not contain the name of a program label, the program stops executing the statements in the DO group and starts executing the first statement after the DO group’s END statement. For example, when the condition in the IF statement in the following program is true (that is, when the value of SUM > 10), the program jumps immediately to the statement following the END statement (in this case, the PUT statement).

```
INIT:
return;

MAIN:
 do while(i<5);
   sum+i;
   i=i+2;
   if (sum>10) then
     do;
        leave;
     end;
   put sum=;
   end;
totalsum=sum;
return;
TERM:
return;
```

Example 2: LEAVE Statements With Label Names In this example, when the condition SUM > 50 is true, the program leaves the LAB1 DO group and returns to the next statement following the DO group (in this case, the PUT statement).

```
INIT:
   sum=45;
return;
MAIN:
   link LAB1;
return;
LAB1:
   do i=1 to 10;
      if (sum>10) then do;
         k=0;
         do until (k>=20);
            sum+k;
            if (sum>50) then leave LAB1;
            k+2;
         end;
```

```
        end;
      end;
      put 'LEAVE LAB1, sum >50 ' sum=;
    return;
    TERM:
    return;
```

See Also

"CONTINUE" on page 300

"DO" on page 356

LEFT

Returns a left-aligned character string

Category: Character

Syntax

lstring=**LEFT**(*string*<,*length*>);

lstring
contains the left-aligned character string. If *lstring* already exists, then specifying a length in the LEFT function affects the current length of *lstring* only if the specified length is less than the trimmed length of the string.
Type: Character

string
is the character string to be left-justified.
Type: Character

length
is the length in which the character string is to be left-justified. The default is the maximum length of *lstring*.
Type: Numeric

Details Any leading blanks in the string are removed so that the first character in the string is nonblank. The default length of the returned value is the trimmed length of the left-aligned string. Use *length* to specify a different maximum length for the returned string.

In order for LEFT to work properly when *lstring* is a window variable, set the justification field (**JUST**) in the field attribute window for *lstring* to **NONE**.

To right-justify a character string, use RIGHT. To center a character string, use CENTER.

See Also

"RIGHT" on page 620

"CENTER" on page 275

LEGEND

Displays a legend window or refreshes the current LEGEND window

Category: Legend

Syntax

CALL LEGEND(<*window-name*<,*back-color*
<,*border-color*<,*border-attr*>>>>);

window-name
is the name that is displayed in the window border. Once assigned, a window name is displayed on subsequent legend windows until it is changed by another LEGEND routine that assigns a different name or that assigns a null string (`''`) to delete the name from the current legend window.
Type: Character

back-color
is a background color name, or `''`for the default color. Available colors are BLACK, BLUE, BROWN, CYAN, GRAY, GREEN, MAGENTA, ORANGE, PINK, RED, WHITE, and YELLOW. SASCOLOR window elements can also be used.
The default background color is the SASCOLOR window element "Secondary Background."
Type: Character

border-color
is a border color name, or `''`for the default color. Available colors are listed under *back-color*. SASCOLOR window elements can also be used for *border-color*.
The default border color is the SASCOLOR window element "Secondary Border."
Type: Character

border-attr
is a border attribute, or `''`for the default attribute. Attributes are NONE, BLINKING, HIGHLIGHT, HIREV, REVERSE, and UNDERLINE. If you specify a SASCOLOR window element for *border-color*, then *border-attr* is ignored, because the SASCOLOR window element contains a display attribute.
The default border attribute is the SASCOLOR window element "Secondary Border".
Type: Character

Details The LEGEND routine displays legend text that has been previously specified with the PUTLEGEND routine. You can specify any combination of optional arguments for LEGEND.

By default, the LEGEND window has the following characteristics:

- The window occupies rows 1 through 6 and columns 1 through the width of the display device.

- □ The window name is either the name that was specified by the last LEGEND routine or the name of the current legend window.

Before invoking the LEGEND routine, you may need to resize the associated application window so that it does not obscure the LEGEND window. To do this, either use the WDEF routine or assign a new size to the window.

Additionally, you can specify a size for a legend window by using the WREGION routine before calling the legend.

Example

Suppose you have two FRAME entries, X and Y. Assume that X.FRAME contains two pushbuttons named PUSHPOP and ENDLGND, and that X.SCL contains the corresponding control labels. When the PUSHPOP button is activated, the PUSHLEGEND call will save X's legend, and the Y.FRAME will be displayed. Y will then set up and display its own legend. After the return from Y, the POPLEGEND call will restore X's legend.

If the ENDLGND button is activated, ENDLEGEND will close the LEGEND window, and the application window will be restored to its original size.

X.SCL contains the following program:

```
INIT:
      /* Get the number of rows and columns for later */
      /* use.                                    */
   nr = winfo('numrows');
   nc = winfo('numcols');
      /* Resize the application window to */
      /* start at row 10. */
   call wdef(10, 1, nr-9, nc);

      /* Set the size of the LEGEND window - row 1  */
      /* through row 9.  Pass a null string */
      /* as the fifth parameter to indicate */
      /* that the LEGEND window has no      */
      /* command area.                      */
   call wregion(1, 1, 9, nc, '');

      /* Set up the legend text and display it. */
   call putlegend(1,'This is line one of the legend for X',
                  'yellow','none');
   call putlegend(2,'This is line two of the legend for X',
                  'yellow','none');
   call legend('Sample LEGEND Window for X',
               'gray','blue');
return;

MAIN:
return;

   /* PUSHPOP label.  If this is executed, */
   /* we'll save the current  */
   /* legend and call y, */
   /* which will display its own legend. */
PUSHPOP:
      /* Push and call. */
   call pushlegend();
   call display('y.frame');

      /* Restore the original legend. */
   call poplegend();
return;

   /* ENDLGND label. If this is executed,    */
   /* the LEGEND window will be              */
   /* closed, and the application window     */
   /* will be restored to its original size. */
ENDLGND:
   call endlegend();
   call wdef(1, 1, nr, nc);
return;

TERM:
return;
```

Y.SCL contains the following program:

```
INIT:
      /* Set up and display Y's own */
      /* LEGEND window. */
   nr = winfo('numrows');
   nc = winfo('numcols');
   call wdef(10, 1, nr-9, nc);
   call wregion(1, 1, 9, nc, '');
   call putlegend(1,'This is line one of the legend for Y',
                  'yellow', 'none');
   call putlegend(2,'This is line two of the legend for Y',
                  'yellow', 'none');
   call legend('Sample LEGEND Window for Y',
               'gray', 'blue');
return;
MAIN:
TERM:
return;
```

See Also

"ENDLEGEND" on page 366

"POPLEGEND" on page 596

"PUSHLEGEND" on page 603

"PUTLEGEND" on page 606

LENGTH

Returns the length of a trimmed character string

Category: Character

Syntax

length=**LENGTH**(*cval*,<'NOTRIM'>);

length
contains the length of the trimmed character string.
Type: Numeric

cval
is the character value whose length is to be determined.
Type: Character

'NOTRIM'
specifies that trailing blanks should be counted as part of the string length.
Type: Character

Details The resulting value is the position of the right-most nonblank character in the specified string *cval*.

When NOTRIM is specified, LENGTH returns the length of a string, including trailing blanks.

By default, variables automatically remove leading blanks when values are assigned to them. The $CHAR format and informat must be assigned to a variable in order for leading blanks to be considered in determining the length of a variable.

Examples

Example 1: Using the LENGTH Function Return the length of the character variable S:

```
length s $ 5;
s='ab ';
l=length(s);
put 'L='l;
```

This program produces the following output:

```
L=2
```

Example 2: Using the LENGTH Function with NOTRIM Return the length of the character variable S, using the NOTRIM option:

```
s = 'xy  ';
l = length(s, 'notrim');
put 'L='l;
```

This program produces the following output:

```
L=4
```

See Also

"MLENGTH" on page 549

LENGTH

Declares variables and specifies their length and whether their data type is numeric or character

Category: Declarative Statement

Comparisons: SAS Statement with limitations in SCL

Syntax

LENGTH<*variable-list*><DEFAULT=*n*>;

variable-list
is one or more variables, specified as *variable-1* <. . . *variable-n*> <$> *length*, where

variable
names a variable to be assigned a length.

$
designates that the preceding variable or variables are character type.

length
is the length of the preceding variable or variables. *Length* can range from 1 to 32,767 for character variables. All numeric variables have a length of 8. If you specify a different length for a numeric variable, SCL still reserves 8 bytes for it.
Type: Character

DEFAULT=*n*
is the maximum length of character variables that are not defined by a LENGTH statement. If this option is not used, the default length for character variables is 200.
Type: Numeric

Details In SCL, LENGTH is a declarative statement and can be used only to set the lengths of nonwindow variables. If you attempt to specify a length for a window variable, a compile error occurs.

You typically place LENGTH statements at the beginning of a program, before the first labeled section. A compiler error occurs if you attempt to place a LENGTH statement within a DO group or within a conditional clause.

You can use the LENGTH statement to reduce the amount of memory required for storing character-type nonwindow variables. For example, if your program assigns only single-character values to the variable CODE, and if the default length of character variables is 200, then you can save 199 bytes of storage space by defining the length of the variable explicitly in the SCL program. To do so, use a statement like the following:

```
length code $ 1;
```

For details about the LENGTH statement in the base SAS language, see *SAS Language Reference: Dictionary* .

Example

Set the maximum length of all character variables that are not defined by a LENGTH statement to 150:

```
length default=150;
length a $ 8;
INIT:
   b='';
   max_a=mlength(a);
   max_b=mlength(b);
   put max_a= max_b=;
```

```
  return;
```

The output is:

```
  max_a=8 max_b=150
```

See Also

"DECLARE" on page 330

LETTER

Displays the FSLETTER window or sends a letter that was created with the FSLETTER procedure

Category: Utility

Syntax

CALL LETTER(*letter-entry*<,*open-mode*<,*table-name*>>);

letter-entry
is a catalog containing one or more LETTER, FORM, or EDPARMS entries. A one- or two-level name is assumed to be *catalog* or *libref.catalog*. The catalog is created if it does not already exist.
Type: Character

open-mode
specifies the type of access to the FSLETTER window:

'BROWSE'
opens the catalog or letter-entry for browsing.

'EDIT'
opens the catalog or letter-entry for editing. (This is the default.)

'PRINT'
prints a letter for each row in the SAS table specified by *table-name*. The SEND window is not displayed for the items that are printed. **PRINT** mode is valid only when the specified entry is a letter.

'SEND'
displays the FSLETTER SEND window for one row (or letter), enabling a user to customize the letter. To use this option, you do not have to specify a value for *table-name*. If a table name is provided, the letter is displayed in the SEND window with the fields filled with values from the first row in the table. This mode is valid only when the specified entry is a letter.
Type: Character

table-name
is the SAS table containing values for the fill-in fields. Use the syntax <*libref.*>*member-name*<(*SAS-data-set-options*)>. If you omit *libref*, the default SAS data library, WORK, is used.
Specify `''`to use the _LAST_ table. If no _LAST_ table exists, the program halts.
You can add a list of SAS data set options following the table name. The list must be enclosed in parentheses. Valid data set options include DROP, KEEP, RENAME,

WHERE, and CNTLLEV. See *SAS Language Reference: Dictionary* for a list of data set options and their descriptions.
Type: Character

Details The LETTER routine displays the FSLETTER window or sends a letter.

Note: The FSLETTER window is not displayed if a PRINT argument is used. △

If the value supplied for *letter-entry* is a three- or four-level name, the user is returned to the calling application when the FSLETTER window is closed. If a one- or two-level name is supplied, the user is returned directly to the calling application when the SAS Explorer window is closed.

SAS data set options can be specified by enclosing them within parentheses immediately following the *table-name* argument, as in the following example:

```
call letter('my.letters.subscrib','print',
    'personal(where=(name="John"))');
```

Examples

- Open the FSLETTER window to edit a document named SUBSCRIB:

  ```
  call letter('my.letters.subscrib');
  ```

- Send a copy of the SUBSCRIB letter for each row in the SAS table SUBSCRIB.DATA. Direct FSLETTER output to a print file when you use CALL LETTER.

  ```
  rc=filename ('myfile',fname);
  call execcmdi('prtfile myfile');
  call letter('my.letters.subscrib','print','subscrib.data');
  ```

- Send a copy of the SUBSCRIB letter for the first row in the SAS table SUBSCRIB.DATA:

  ```
  call letter('my.letters.subscrib','send','subscrib.data');
  ```

SEND mode for the letter SUBSCRIB accepts user input.

See Also

"FSEDIT" on page 433

LIBLIST

Displays a host selection window that lists the currently assigned librefs, and returns user's selections

Category: SAS Table

Syntax

selections=**LIBLIST**(<*sel-excl*<,*engine*<,*message* <,*autoclose*<,*num-sel*>>>>>);

selections
contains one or more librefs from the list, or blank if no selection is made. Multiple selections are separated by blanks. By default, *selections* is 200 bytes long. To

accommodate values longer than 200 bytes, explicitly declare *selections* with a longer length.
Type: Character

sel-excl
is one or more librefs to include or exclude from the selection list window. Specify names using a style described in “Styles of Name Specification” on page 514.
Type: Character

engine
is one or more engines to use as criteria for determining which librefs are displayed. Specify names using a style described in “Styles of Name Specification” on page 514.
Type: Character

message
is the text for a message to display above the selection list. The default message tells users to make up to the number of selections in *num-sel*.
Type: Character

autoclose
is an obsolete argument but is retained for compatibility with earlier releases. If you want to specify a value for *num-sel*, then specify `''`as a placeholder for this argument.
Type: Character

num-sel
specifies the maximum number of items a user can select from the list. To display the list for information purposes only (no selections allowed), specify 0. To specify unlimited selections, use a value such as 9999 that is larger than the number of available selections. A user cannot make a number of selections that exceeds the number of items in the list.
Type: Numeric

Styles of Name Specification

To specify more than one name, separate the names with a space — for example, **`MYLIB1 MYLIB2`**.

To specify all names, use an asterisk (`'*'`) or a null string (`''`).

To specify all names *except* those listed after the NOT sign, use a NOT sign (¬ or ^) followed by one or more names. For example, **`^MYLIB1`** displays all defined librefs except **`MYLIB1`**.

Details LIBLIST opens the Library Selector window, which lists librefs, engines, and the physical names of the operating system files. However, only the selected libref is returned. The window contains a **`Browse`** button which a user can select to display the SAS Explorer window and select from the librefs that are currently assigned and their contents. After browsing data libraries, a user can select

File ▶ Close

to return to the Library Selector window.

If you omit all the arguments for LIBLIST (for example, **`selections=liblist();`**), the selection list window lists all librefs that have been assigned in the current SAS session.

You can provide a default value that will be initially selected when the libref selection list is displayed. To do this, assign the value to the *selections* variable before calling LIBLIST.

If a user closes the selection list window without making a selection, LIBLIST returns a blank value unless there was an initial value for *selections* before LIBLIST was called.

Selections from the window can be returned in the current result list, if one is available. The current result list is a special SCL list that is automatically filled with the values that are selected from a selection list. To use a current result list, use the MAKELIST function to create the list, and use the CURLIST function to designate it as the current result list. The current result list must exist before you call the LIBLIST function.

When LIBLIST is invoked, the current result list is cleared. After LIBLIST is invoked, the result list contains the following named items:

TAG
: identifies the list as one that was created by the LIBLIST function.
  Type: Character

COUNT
: contains the number of selected librefs, or 0 if a user makes no selections or issues a CANCEL command in the list window.
  Type: Numeric

LIBREF
: contains the name of each selected libref. There is one LIBREF element for each selected libref name.
  Type: Character

LIBNAME
: contains the physical name of the operating system file for each selected libref. There is one LIBNAME element for each selected libref.
  Type: Character

Examples

- Create a selection list that displays all librefs except MYLIB1 and MYLIB2, and display the message **'Choose a libref'**.

```
select=liblist('^mylib mylib2','*',Choose a libref');
```

- Create a selection list that displays all librefs associated with the V609 engine, and exclude the librefs SASHELP and MAPS. Allow users to make up to three selections.

```
select=liblist('^sashelp maps','v609',
'Choose up to 3 librefs','',3);
```

- Create a current result list to receive user selections. Use MAKELIST to create the list and CURLIST to define it as the current result list. Display all librefs except MYLIB1 and MYLIB2, and allow users to make up to five selections. Use a DO loop to retrieve the selections from the current result list.

```
listid=makelist();
rc=curlist(listid);
select=liblist('^ mylib1 mylib2',' ',
'Choose up to 5 librefs','', 5);
n=getnitemn(listid,'COUNT');
do i=1 to n;
   libref=getnitemc(listid,'LIBREF',i);
   physname=getnitemc(listid,'LIBNAME',i);
   put libref= physname=;
end;
```

See Also

"CATLIST" on page 269

"DIRLIST" on page 347

"FILELIST" on page 401

LIBNAME

Assigns or deassigns a libref for a SAS data library

Category: SAS Table

Syntax

sysrc=**LIBNAME**(*libref*<,*SAS-data-library*<,*engine* <,*options*>>>);

sysrc
contains the return code for the operation:

0	The operation was successful.
>0	The operation was not successful.
<0	The operation was completed, but a warning or a note was generated.

Type: Numeric

libref
is the libref to assign.
Type: Character

SAS-data-library
is the physical name of the SAS data library to be associated with the libref. This name can be up to 32K characters long. Specify this name as required by the host operating system.
Type: Character

engine
is the engine to use for accessing the SAS files in the data library. If you are specifying a SAS/SHARE server, the engine should be REMOTE.
Type: Character

options
are options that are used by the specified engine. For information about engines and options, see the SAS documentation for your operating environment.
Type: Character

Details The LIBNAME function follows the rules for the LIBNAME statement in base SAS software.

Operating Environment Information: Some operating systems allow a *SAS-data-library* value of `' '`(with a space) and some allow `'.'` to assign a libref to the current directory. The behavior of LIBNAME when a single space or a period is specified for *SAS-data-library* is host dependent. Under some operating systems, librefs can be assigned using system commands outside the SAS session. △

Examples

Example 1: Assigning a Libref Assign the libref NEW to the SAS data library TEST.DATA. If an error or warning occurs, display the message on the message line.

```
if (libname('new','test.data'))then
   _msg_=sysmsg();
```

Example 2: Deassigning a Libref Deassign the libref NEW. If an error or warning occurs, display the message on the message line.

```
if (libname('new')) then
   _msg_=sysmsg();
```

Example 3: Using a List to Assign a Libref to Multiple SAS Data Libraries Assign the libref DEF to several PC files whose names are stored in an SCL list:

```
lid=makelist();
rc=insertc(lid,''(`',-1);
rc=insertc(lid,'''M:\SAS\MAPS''',-1);
rc=insertc(lid,'''C:\CATALOGS\sasuser''',-1);
rc=insertc(lid,'')'',-1);
rc=libname('DEF',' ','','',lid);
```

Assign the libref DEF to several UNIX files whose names are stored in an SCL list:

```
v1=''(/mylib/store/data/facilities''';
v2='''/mylib/store/data/hresorces''';
v3='''/mylib/store/data/supplies')'';
lid = makelist ();
rc =insertc(lid,v1,-1);
rc =insertc(lid,v2,-1);
rc =insertc(lid,v3,-1);
RC =LIBNAME('DEF',' ','','',lid);
```

See Also

"LIBREF" on page 517

LIBREF

Verifies that a libref has been assigned

Category: SAS Table and Utility

Syntax

sysrc=**LIBREF**(*libref*);

sysrc
contains the return code for the operation:

=0	The operation was successful.

<0	The operation was completed, but a warning or a note was generated.
>0	The operation was not successful.

Type: Numeric

libref
is the libref to be verified.
Type: Character

Example

Verify a libref. If an error or warning occurs, the message is displayed on the application window's message line.

```
if (libref('sashelp'))
then _msg_=sysmsg();
```

See Also

"LIBNAME" on page 516

LISTC and LISTN

Display a selection list window containing values stored in a catalog entry

Category: Selection List

Syntax

selections=**LISTC**(*entry*<,*message*<,*autoclose*<,*num-sel*>>>);

selections=**LISTN**(*entry*<,*message*<,*autoclose*<,*num-sel*>>>);

selections
contains one or more character values that have been selected by the user.
For LISTC, if a selection is not made, *selections* will be blank. Multiple selections are separated by blanks. By default, *selections* is 200 bytes long. To accommodate values longer than 200 bytes, explicitly declare *selections* with a longer length.
For LISTN, *selections* is the first value that the user selected. The value is numeric.
Type: Character or Numeric

entry
is a LIST entry (for LISTN) or a HELP, LIST, or MENU entry (for LISTC). The entry must be specified as *entry.type* for an entry in the current catalog or as *libref.catalog.entry.type* for an entry in a different catalog.
Type: Character

message
is text for a message to be displayed above the selection list. The default message tells users to make up to the number of selections specified by *num-sel*, or 1 if *num-sel* is not provided. The default is 1.
Type: Character

autoclose
specifies whether the selection list window closes automatically after a user makes a selection when only one choice is allowed:

'Y'	closes the window automatically. (This is the default.)
'N'	leaves the window open until the user explicitly closes it.

This option is ignored when *num-sel* is not 1. However, use `''`as a placeholder if you are also specifying a value for *num-sel*.
Type: Character

num-sel
specifies the maximum number of items a user can select from the list. To display the list for information purposes only (no selections allowed), specify 0. To specify an unlimited number of selections, use a value such as 9999 that is larger than the number of available selections. The default is one selection.
Type: Numeric

Details LISTC automatically displays a selection list containing character values that are stored in a LIST, HELP, or MENU entry. A LIST entry that is used with LISTC must be of character type. Typically, a LIST entry is used if the selections in the LIST entry are self-explanatory. A HELP or MENU entry is used if a definition is needed next to the selection.

LISTN automatically displays a selection list containing numeric values stored in a LIST entry, which must be of numeric type. The numeric values are displayed using the format that was specified for the LIST entry. If no format was specified, the values are displayed using the BEST. format.

For a selection list that is produced with a LIST entry, you can provide a default or initial selected value by specifying a value for *selections* before calling LISTC. If *selections* contains valid values when LISTC is invoked, those values are automatically designated as selected when the selection list is displayed.

When multiple selections are allowed in LISTN, *selections* contains the first value selected from the list. However, the values for all selections can be returned in the current result list, if one is available. The current result list is a special SCL list that is automatically filled with the values selected from a selection list. To use a current result list, use the MAKELIST function to create the list, and use the CURLIST function to designate it as the current result list. The current result list must exist before you call LISTC. You can use GETITEMC to retrieve values from the list.

Examples

Example 1: Using LISTC with a LIST Entry Open the entry MYLIST.LIST in the current catalog, and then display it as a selection list. Users can make up to four selections. The selected values are retrieved from the current environment list.

```
listid=makelist();
rc=curlist(listid);
selections=listc('mylist.list','','n',4);
n=listlen(listid);
do i=1 to n;
   item=getitemc(listid,i);
   put item=;
end;
```

Example 2: Using LISTC with the Current Result List Create LIST_C and make it the current list. Use LISTC to display a selection list containing the values ABC, DEF, GHI, and JLK, which are stored in MYCHAR.LIST, and allow a user to make up to 4 selections.

```
list_c=makelist();
cur_list=curlist(list_c);
   /* Display the list and put the user     */
   /* selection in SELECTIONS.              */
   /* Then print the number of selections.  */
selections=listc('mychar.list',' ',' ',4);
put 'User selected' selections;
   /*  Find out the number of items          */
   /*  in LIST_C and print the number.       */
num_selected=listlen(list_c);
put 'Total number selected is' num_selected;
   /* Get the selections from               */
   /* the current list                      */
   /* and print each one.                   */
do i=1 to num_selected;
   item=getitemc(list_c,i);
   put 'Item' i 'is ' item;
end;
```

Testing the program and selecting GHI, DEF, JKL, and then ABC produces the following output:

```
User selected GHI DEF JKL ABC
Total number selected is 4
Item 1 is GHI
Item 2 is DEF
Item 3 is JKL
Item 4 is ABC
```

Example 3: Using LISTN with the Current Result List Create LIST_N and make it the current list. Use LISTN to display a selection list containing the numbers 1, 2, 3, and 4, which are stored in MYLIST.LIST, and allow a user to make up to 4 selections.

```
list_n=makelist();
cur_list=curlist(list_n);
   /* Display the list and put the first user   */
   /* selection in SELECTED_FIRST,              */
   /* then print the number of user selections. */
selected_first=listn('mylist.list',' ',' ',4);
put 'First selection is ' selected_first;
   /*  Find out the number of items in LIST-N  */
   /* and print the number.                    */
num_selected=listlen(list_n);
put 'Total number selected is ' num_selected;
   /* Get any other selections from           */
   /* the current list                        */
   /* and print each number.                  */
do i=1 to num_selected;
   item=getitemn(list_n,i);
   put 'Item ' i 'is ' item;
end;
```

Testing the program and selecting 3, 2, 4, and 1, produces the following output:

```
First selection is 3
Total number selected is 4
Item 1 is 3
Item 2 is 2
Item 3 is 4
Item 4 is 1
```

See Also

"DATALISTC and DATALISTN" on page 326

LISTLEN

Reports the length of an SCL list

Category: List

Syntax

n=**LISTLEN**(*list-id*);

n

contains either the length of an SCL list or status information:

>0	the length of a non-empty list
0	the list is empty
–1	the list identifier is invalid.

Type: Numeric

list-id

is the identifier of the list whose length is being queried, or any other number.

Type: Numeric or List

Details The length of a list is the number of items in the list, excluding the contents of sublists. Because LISTLEN returns -1 if *list-id* is an invalid list identifier, you can use LISTLEN to determine whether a list exists. For example:

```
listid=getniteml(envlist('G'),'MYLIST');
invalid=(listlen(listid)=-1);
if invalid then
do;
   put 'MYLIST in the global environment has been deleted.';
   stop;
end;
```

Example

Create the empty list LISTA, and then insert LISTA into a copy of itself, LISTB. The lengths of the two lists are then computed and are stored in the variables LEN_A and LEN_B.

```
lista=makelist();
listb=copylist(lista);
listb=insertl(listb,lista);
len_a=listlen(lista);
len_b=listlen(listb);
_msg_='The length of LISTA is '||len_a||' and '||
      'the length of LISTB is '||len_b;
```

This example shows that the length of LISTA is 0, whereas the length of LISTB is 1.

See Also

"MAKELIST" on page 535

"MAKENLIST" on page 536

LNAMECHK

Validates a path string

Category: Image

Syntax

rc= **LNAMECHK**(*path-string*);

rc
contains the return code for the operation:

0	The path string is a valid path to a file.
>0	The path string is not a valid path to a file.

Type: Numeric

path-string
is the string generated by LNAMEMK.
Type: Character

Details LNAMECHK validates that the specified path string refers to an external file that exists. It does not determine whether the file contains a readable image.

Example

Test whether a file exists:

```
imgpath=lnamemk(2,fromdir,file)
rc=lnamechk(2,imgpath);
if (rc ne 0) then
   do;
      _msg_="File does not exist.";
   end;
```

LNAMEGET

Decodes a path string

Category: Image

Syntax

rc=**LNAMEGET**(*path-string*,*type*,<*name-string1*<,*name-string2*><,*options*>>);

rc
contains the return code for the operation:

0	successful
>0	not successful

Type: Numeric

path-string
is the string generated by LNAMEMK.
Type: Character

type
is the type of the path. See “LNAMEMK” on page 524 for more information. If no other arguments are specified, the function returns only the type.
Type: Numeric

name-string
is the *name-string* provided in LNAMEMK. Specify enough *name-string* arguments for the *type*. See “LNAMEMK” on page 524 for more information.
Type: Character

options
are any options used with LNAMEMK.
Type: Character

Details Path strings that are created by LNAMEMK are not readable, and their internal format may change from release to release. The *only* way to decode a path string is to use LNAMEGET.

You may find it useful to encode an image filename with LNAMEMK and to store that path string in a SAS data set. Then, later retrieve the path string and use LNAMEGET to find the arguments that were originally specified in LNAMEMK.

If you use the *type*, *name-string*, and *options* arguments, they are filled with the corresponding arguments specified in LNAMEMK (such as the libref/member name, physical pathname, and so on). The number of optional arguments that you specify must match the number specified in LNAMEMK.

Examples

- Encode and decode a pathname. Store the type of path in TYPE, and store the pathname in IMGFILE.

```
imgpath=lnamemk(1,filename);
rc=lnameget(imgpath,type,imgfile);
```

- Encode and decode the location of an image file. Store the directory that contains the image file in DIR, and store the filename in IMGFILE.

```
imgpath=lnamemk(2,dirname,filename);
rc=lnameget(imgpath,type,dir,imgfile);
```

See Also

"LNAMECHK" on page 522

"LNAMEMK" on page 524

LNAMEMK

Makes a path string for an image file

Category: Image

Syntax

path-string=**LNAMEMK**(*type*,<*name-string1*<,*name-string2*><,*attributes*>>);

path-string
contains a packed string containing information about the file path and format. Declare a length of at least 300 characters.
Type: Character

type
is a number from 1 through 5 that specifies the type of path used to read the external file.
Type: Numeric

name-string
is the string that identifies the location of the image. The specification depends on the value specified for *type*.

For type	use name-string1	and name-string2
1	physical-pathname	
2	directory-pathname	filename
3	fileref	
4	fileref	filename
5	libref.catalog.member	

Type: Character

attributes
specify file-specific attributes. See Table A2.1 on page 773and Table A2.2 on page 775 for possible choices. The FORMAT= attribute must be specified for Targa images, for images residing in SAS catalogs, and for host-specific formats. FORMAT is not required in other cases, but it is always more efficient to specify it.
Type: Character

Details LNAMEMK creates a character variable that contains information about the location of the image as well as other image attributes.

The path string can be used with the READ and WRITE commands in IMGOP or with the image object. The path string contains binary data and can be decoded only with the LNAMEGET and LNAMECHK functions.

Example

Create path strings for image files:

```
length file $ 200;

file=lnamemk(1,filename,'format=gif');
file=lnamemk(2,directory,filename,'format=gif');
file=lnamemk(3,fileref,'format=gif');
file=lnamemk(4,fileref,filename,'format=gif');

imgentry=libref||"."||catalog||"."||member;
file=lnamemk(5,imgentry,'format=cat');
```

See Also

"IMGOP" on page 472

"LNAMECHK" on page 522

"LNAMEGET" on page 523

LOADCLASS

Loads a class and returns its identifier number

Category: Object Oriented

Syntax

class-id=**LOADCLASS**(*class-name*);

class-id
contains the identifier that has been assigned to the class. If the class is not loaded, *class-id* contains 0.
Type: Numeric

class-name
is the one- to four-level name of the CLASS catalog entry to load. If *class-name* is a one- or two-level name, then the current search path is used to find the CLASS entry. If the CLASS entry was previously loaded, then the same class identifier is returned. Otherwise, the CLASS entry is loaded from the catalog into the application class list, and the class identifier is returned in *class-id*.
Type: Character

Details LOADCLASS loads a class definition from a CLASS catalog entry. The identifier number that LOADCLASS returns can be used to create an instance of the class with the INSTANCE function.

Example

Load SASUSER.CLASSES.TIMER.CLASS and use the INSTANCE function to create an instance of the TIMER class:

```
timerclass=loadclass('sasuser.classes.timer');
timer=instance(timerclass);
```

See Also

"APPLY" on page 252

"INSTANCE" on page 485

"LOADRES" on page 526

"_NEO_" on page 559

"NOTIFY" on page 572

"SEND" on page 644

"SUPAPPLY" on page 679

"SUPER" on page 681

LOADRES

Loads a RESOURCE entry

Category: Object Oriented

Syntax

resource-id=**LOADRES**(*resource-name*);

resource-id
contains the identifier that is assigned to the resource list.
Type: Numeric

resource-name
is the RESOURCE catalog entry to load. If *resource-name* is a one- or two-level name, the current search path is used to find the RESOURCE entry.
Type: Character

Details LOADRES loads a list of classes from a RESOURCE entry. This list is called a resource list. RESOURCE entries are used primarily by FRAME entries, although you can create RESOURCE entries for component classes as well. This function is useful for loading several classes or even entire class hierarchies at one time instead of having to load several CLASS entries.

If a class contained in the resource list has already been loaded, the existing class replaces the class in the resource list (although the RESOURCE entry is not modified). This prevents duplicate class lists for the same class name.

Example

Load a resource list that is stored in APPQR.HIER1.GROUPS.RESOURCE, then load several classes contained in the RESOURCE entry. After the LOADRES call, the LOADCLASS calls do not have to read the classes from the catalog.

```
groups = loadres('appqr.hier1.groups.resource');
   c1=loadclass('appqr.hier1.c1.class');
   c2=loadclass('appqr.hier1.c2.class');
   c3=loadclass('appqr.hier1.c3.class');
```

See Also

"APPLY" on page 252

"INSTANCE" on page 485

"LOADCLASS" on page 525

"_NEO_" on page 559

"NOTIFY" on page 572

"SEND" on page 644

"SUPAPPLY" on page 679

"SUPER" on page 681

LOCATEC and LOCATEN

Search a SAS table for a row that contains a specified value

Category: SAS Table

Syntax

rc=**LOCATEC**(*table-id*,*col-num*,*cval*<,*sort*<,*direction*>>);

rc=**LOCATEN**(*table-id*, *col-num*,*nval* <,*sort*<,*direction*>>);

rc
contains information about the search:

>0	the number of rows read before a match is found
0	no row with a matching value was found

Type: Numeric

table-id
is the identifier that was assigned when the SAS table was opened. If *table-id* is invalid, the program halts.
Type: Numeric

col-num
is the number of the column to search for. This number can be returned by the VARNUM function. If the number is invalid, the program halts and sends a message to the log.
Type: Numeric

cval
is the character value for LOCATEC to search for. If *cval* is not a character value, the program halts and sends a message to the log.
Type: Character

nval
is the numeric value for LOCATEN to search for. If *nval* is not a numeric value, the program halts and sends a message to the log.
Type: Numeric

sort
indicates whether the SAS table is sorted:

'A'	The table is sorted in ascending order.
'D'	The table is sorted in descending order.
'U'	The table is not sorted. (This is the default.)

Type: Character

direction
specifies the direction in which to search the SAS table:

'A'	searches all rows, starting with the first row. (This is the default.)
'B'	searches from the previous row backward.
'F'	searches from the next row forward.

Type: Character

Details LOCATEC and LOCATEN do not search for partial values. For LOCATEC, preceding blanks are part of *cval* but trailing blanks are not. Therefore, you can facilitate searching for LOCATEC by using the LEFT function to left-justify character values.

LOCATEC and LOCATEN search all rows, starting with the first row by default and skipping rows marked for deletion. When a WHERE clause is active, these functions search the rows that meet the WHERE condition for a match. If a matching row is found, it is loaded into the Table Data Vector (TDV). Otherwise, the current row remains in the TDV.

LOCATEC and LOCATEN return the number of rows read before a match is found. This number may not correspond to the row number where the match is found because these functions skip deleted rows. Moreover, if a WHERE clause is active, they read only the rows, including any appended rows, that meet the WHERE condition. Also, if *direction* is supplied, the number returned is the number of rows read from the previous row where the search began. By default, the search direction is forward, starting with the first row in the table.

If the table is sorted, then specifying **'A'** or **'D'** for *sort* uses the more efficient binary search algorithm. Perform a binary search only when you have member-level access so that no one else can be editing the table concurrently. With a binary search, LOCATEC and LOCATEN make assumptions about how the data is sorted, and they assume that they can identify the first and last rows. If the table is being edited concurrently, rows could be appended so that the table is no longer in sorted order. As a result, the binary search might not find the correct values.

Examples

Example 1: Using the LOCATEC Function Locate a customer named SMITH in the PAYROLL table. The table is opened with a *table-id* of TABLEID and is sorted by NAME. The customer's name is specified in the CUSTOMER column.

```
customer='SMITH';
rc=locatec(tableid,varnum(tableid,'name'),customer,'a');
if (rc=0) then _msg_=
   'There is no customer named '||customer||'.';
else do;
      ...more SCL statements...
     end;
return;
```

Example 2: Using the LOCATEN Function Locate a house whose price is $94,000 in the SASUSER.HOUSES table, which is opened with a *table-id* of HOUSEID. The price is specified in the window variable PRICE.

```
houseid=open('sasuser.houses');
price=94000;
rc=locaten(houseid,varnum(houseid,'price'),price);
if (rc=0) then
   _msg_='No house is priced at '||
          putn(price,'dollar9.2')||'.';
else do;
   rows=curobs(houseid);
   _msg_=
      'The specified price was found in row '||rows;
   end;
return;
```

See Also

"FETCH" on page 389

"FETCHOBS" on page 390

"GETVARC and GETVARN" on page 450

"SET" on page 646

LOCK

Locks or unlocks a SAS table or a SAS catalog entry

Category: SAS Table

Syntax

sysrc=**LOCK**(*member*<,*action*>);

sysrc
contains the return code for the operation:

0	successful
>0	not successful
<0	the operation was completed, but a warning or a note was generated.

member
is a member of a SAS data library or a SAS catalog entry. The value that you specify can be a one-, two-, three-, or four-level name. A one-level name is presumed to be a libref, whereas a two-level name defaults to the SAS table type DATA.
Type: Character

action
specifies an action to be performed on the SAS table or catalog entry:

'CLEAR'
unlocks the specified SAS table(s) or SAS catalog entry.

'LOCK'
locks the specified SAS table(s) or SAS catalog entry. (This is the default.)

'QUERY'
queries the lock status of a SAS table or a SAS catalog entry.

_SWNOLKH
not currently locked. SYSRC of -630099.

_SWLKUSR
locked by another user. SYSRC of -630097.

_SWLKYOU
locked or in use by the caller. SYSRC of -630098.

Type: Character

Details If *action* is not provided, the action defaults to LOCK.

Example

Lock the data library that is associated with a libref of A, unlock data view LIB.A, and lock LIB.A.B.PROGRAM. Then, query the lock state of the FOO.ONE table:

```
rc=lock('a');
rc=lock('lib.a.view','clear');
rc=lock('lib.a.b.program');
rc=lock('foo.one.data','query');
if (rc=%sysrc(_SWLKUSR)) then
  _msg_='Table foo.one is currently locked.';
```

LOOKUPC

Searches for a string among a list of valid tokens

Category: Command

Syntax

rc=**LOOKUPC**(*string,token-1<, . . .,token-12>*);

rc
contains the return code for the operation:

0	indicates that no match was found.

>0	is the position in the token list if a unique match was found.
<0	is the negative of the position in the token list if a duplicate token was found.

Type: Numeric

string
is the character value to search for.
Type: Character

token
is up to 12 character values, separated by commas.
Type: Character

Details A *token* can be a name, a literal, digits, or special characters. This function is useful for looking up valid commands.

The function accepts abbreviations as valid commands. That is, the function reports a match if the search string matches the starting characters of a token.

LOOKUPC does not search the token list for embedded strings. For example, the search string LIST would not be found in the token NEWLIST.

Example

Get the command (SURVEY, NEWLIST, or ADDNAME) that the user issued from the command line, and execute code accordingly:

```
array cmds{*} $8 ('SURVEY','NEWLIST','ADDNAME');
INIT:
   control always;
return;
MAIN:
cmdword=WORD(1,'u');
cmdnum=lookupc(cmdword,cmds{1},cmds{2},cmds{3});
select;
when (cmdnum=1)
    ...SCL statements to process SURVEY command...
when (cmdnum=2)
    ...SCL statements to process NEWLIST command...
when (cmdnum=3)
   ...SCL statements to process ADDNAME command...
otherwise _msg_='Command conflict';
   end;
```

In this example, SUR, NEWL, and ADDN are considered valid commands.

LVARLEVEL

Fills an SCL list with the unique values of a column from a SAS table

Category: List

Syntax

rc=**LVARLEVEL**(*dsid*,*varname*,*n-level*<,*list-id*>);

rc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

dsid
is the identifier that was assigned when the table was opened. An invalid *dsid* produces an error condition.
Type: Numeric

varname
is the column whose unique formatted values are to be reported.
Type: Character

n-level
is the name of the column in which the function stores the number of unique values, or levels. This column must be initialized to a nonmissing value before its value is set by LVARLEVEL.

Note: This parameter is an update parameter. See "Input, Output, and Update Parameters" on page 34 for more information. △
Type: Numeric

list-id
is the identifier of the list to fill with the unique formatted values. If *list-id* is not provided, the values are placed in the current result list. An invalid *list-id* produces an error condition.
Type: Numeric or List

Details The values are placed in the list identified by *list-id*, or in the current result list identified by CURLIST (if *list-id* is not specified). The values placed in the list are always character values. It is an error if *list-id* is omitted and you have not created a current result list with the CURLIST function. *n-level* must be a column, because LVARLEVEL uses it to store the number of unique values it finds. *n-level* must be initialized to any value except missing before LVARLEVEL executes.

Examples

Example 1: Placing Values in the Current List Get the unique formatted values for the table column NAME from SASUSER.CLASS, place the values in the current list, and print them:

```
dsid=open('sasuser.class');
nlevels=0;
rc=curlist(makelist());
rc=lvarlevel(dsid,'name',nlevels);
put nlevels=;
call putlist(curlist(),'levels',0);
rc=close(dsid);
```

Example 2: Placing Values in a Specified List Get the unique formatted values for the table column NAME from SASUSER.CLASS, place the values in the specified list, and print them:

```
dsid=open('sasuser.class');
nlevels=0;
listid=makelist();
rc=lvarlevel(dsid,'name',nlevels,listid);
put nlevels=;
call putlist(listid,'levels',0);
rc=close(dsid);
rc=dellist(listid);
```

See Also

"CURLIST" on page 320

"OPEN" on page 576

"VARLEVEL" on page 708

"VARSTAT" on page 714

MAKEARRAY

Creates an array of the given size with all elements in the array initialized to missing for numeric values or blank for character values

Category: Array

Syntax

array=**MAKEARRAY**(*dim1*<,...,*dimN*>);

array
is the dynamic array to be created. A non-dynamic array causes an error condition.
Type: Array

dim1,...,dimN
is the size of each specified dimension. If you specify negative sizes or an invalid number of dimensions, an error condition occurs.
Type: Numeric

Details Unlike static arrays, whose bounds must be set at compile time, you can create and resize (change the bounds of) dynamic arrays at run time. The low bound of the dynamic array will always be 1, and the high bound will be determined as given at run time. If you create a one-dimensional dynamic array with 5 elements, then the low bound and high bound will be 1 and 5, respectively. The array must be declared using an asterisk (*) for the array dimensions with no array elements or initial values specified. The syntax is the same as for a reference array. For example, the following lines declare a one-dimensional numeric dynamic array named A and a two-dimensional character dynamic array named B:

```
DCL num A(*);
DCL char B(*,*);
```

The MAKEARRAY function creates an array of the given size. All elements in the array initialized to missing for numeric values or blank for character values. The number of dimensions must be the same as what was specified in the DECLARE statement.

If you use the MAKEARRAY function to resize a dynamic array, all the data is lost and becomes garbage. If you try to reference an array element without first creating the array, an error occurs.

Dynamic arrays can be used with the other existing array functions (DIM, HBOUND, LBOUND) as long as the array has been created with MAKEARRAY. If you try to use these other functions without first creating the array, a program halt occurs.

Examples

Example 1: Create a One–Dimensional Array This example creates a one-dimensional array of 5 elements.

```
DCL num a(*);
a = makearray(5);
```

Example 2: Create a Two–Dimensional Array This example creates a two-dimensional 5x5 array.

```
DCL num a(*,*);
a = makearray(5,5);
```

Example 3: Create an Array from a Table This example uses table **`work.a`**, which has only numerical variables. The data from all the table rows is placed into a two-dimensional array. The dimensions and size of the array are determined by the number of rows and columns in the table.

```
init:

/* Open the table and create the array. */

DCL num arr(*,*) rc;
dsid = open('work.a');
nlobs = attrn(dsid, 'NLOBS');
nvars = attrn(dsid, 'NVARS');
arr = makearray(nlobs,nvars);

/* Move the contents of the table into the array. */

do i = 1 to dim(arr, 1);
 rc = fetch(dsid);
   do j = 1 to dim(arr, 2);
     arr[i,j] = getvarn(dsid, j);
   end;
 end;

/* Close the table and delete the array. */
call close(dsid);
rc = delarray(arr);
return;
```

See Also

"DECLARE" on page 330

"DELARRAY" on page 332

"REDIM" on page 610

DIM, HBOUND, and LBOUND in *SAS Language Reference: Dictionary*

Chapter 4, "SCL Arrays," on page 37

MAKELIST

Creates an SCL list

Category: List

Syntax

list-id=**MAKELIST**(<*n*<, *visibility*>>);

list-id
contains the identifier of the new list, or 0 if the list could not be created.
Type: Numeric or List

n
is the number of items to place in the list initially. By default, *n* is 0.
Type: Numeric

visibility
specifies whether the list is global or local:

'G'	The list is global and can be shared by all applications executing in the same SAS session. A global list is deleted when the SAS session ends.
'L'	The list is local to the current SAS application. A local list is deleted when the application ends. (This is the default.)

Type: Character

Details MAKELIST creates either an empty list or a list with the number of items specified in *n*. Each item contains a missing value. Use the list identifier returned by MAKELIST with all other SCL list functions that use the list.

SCL lists can contain numeric items, character items, other list items, or object items. Each item can have a name. Both lists and list items have attributes. See Chapter 5, "SCL Lists," on page 47 for complete information about using SCL lists.

Example

Create lists in the local and global environments:

```
n = 12;
   /*  Make an empty local list.  */
list1=makelist();
   /*  Make a local list with 24 items.  */
```

```
list2=makelist(2*n);
   /*  Make an empty global list.  */
list3=makelist(0,'G');
```

See Also

"CLEARLIST" on page 290

"COPYLIST" on page 311

"DELLIST" on page 336

"LISTLEN" on page 521

"MAKENLIST" on page 536

"PUTLIST" on page 607

"SAVELIST" on page 629

MAKENLIST

Creates an SCL list that contains named items

Category: List

Syntax

list-id=**MAKENLIST**(*visibility*,*name-1*<, . . . ,*name-n*>);

list-id
contains the identifier of the new list, or 0 if the list could not be created.
Type: Numeric or List

visibility
specifies whether the list is global or local:

'G'	The list is global and can be shared by all applications executing in the same SAS session. A global list is deleted when the SAS session ends.
'L'	The list is local to the current SAS application. A local list is deleted when the application ends. (This is the default.)

Type: Character

name
is one or more list item names, separated by commas. Item names are converted to uppercase, and trailing blanks are removed. Each name can be any SCL string. The same name can be used more than once.
Type: Character

Details MAKENLIST creates a list that contains an item for each name that you specify. Each item contains a missing value. Use the list identifier returned by MAKENLIST with all remaining functions that manipulate the list. When you create a list of named items, you can assign or access list values by their names as well as by their positions. However, it is more efficient to access items by position rather than by name.

You can use MAKENLIST to create structures that group related information into one list. For example, a row in a SAS table can be placed in a named list where each named item corresponds to the table column of the same name.

Note that the *visibility* argument (L or G) is required and is the first argument, unlike the MAKELIST function. Note also that this function does not use an *n* argument.

Using MAKENLIST is simpler than using MAKELIST and then naming each item independently.

Example

The following statement creates a list of four named items:

```
mylist=makenlist('L','A','B','C');
```

It is equivalent to these four statements:

```
mylist=makelist(3,'L');
rc=nameitem(mylist,1,'A');
rc=nameitem(mylist,2,'B');
rc=nameitem(mylist,3,'C');
```

See Also

"LISTLEN" on page 521

"MAKELIST" on page 535

"NAMEDITEM" on page 555

"NAMEITEM" on page 558

MESSAGEBOX

Displays a host message window with a specified text and icon

Category: Utility

Syntax

text=**MESSAGEBOX**(*textlist-id*<,*icon*<,*buttons*
<,*caption*<,*default*<,*right*>>>>>);

text
contains the text of the button that a user pressed in the message dialog window. This text can be OK, CANCEL, ABORT, RETRY, IGNORE, YES, or NO. When a user presses **`Enter`** instead of selecting a button, either *default* is returned (if specified) or the text of the first button in the message window is returned.
Type: Character

textlist-id
contains the identifier for the SCL list that contains the lines of text to display in the message window. Lines that are too long are wrapped. For example, if there are two lines and the first is too long, the text displays as three lines.
Type: List

icon
specifies the icon to display in the message window:

'I'	Information or note icon (default)
'?'	Query icon
'!'	Warning icon
'S'	Error icon (stop sign/hand)

Type: Character

buttons
specifies the set of command buttons to display in the message window:

'O'	Ok (default)
'OC'	Ok Cancel
'YN'	Yes No
'YNC'	Yes No Cancel
'ARI'	Abort Retry Ignore
'RC'	Retry Cancel

Type: Character

caption
is the title for the message window.
Type: Character

default
is a single character that corresponds to one of the characters specified in *buttons*. *Default* specifies the value that is returned when a user presses **Enter** in the message window instead of selecting a button. If *default* is not supplied, the default selection is the text of the first button in the message window.
Type: Character

right
specifies whether the text in the message window is right- or left- justified:

'N'	Left justify the text. (default)
'Y'	Right justify the text.

Type: Character

Details MESSAGEBOX calls a host message window and specifies text to be displayed there. It can also specify an icon, one or more buttons, and a title to display in the window.

If the message window cannot open, or if *textlist-id* is invalid, the program halts. Otherwise, MESSAGEBOX returns OK, CANCEL, ABORT, RETRY, IGNORE, YES, or NO. On hosts that allow users to close the message window without selecting a button, CANCEL is returned even if it is not one of the button choices.

Example

Create a requestor window to prompt users to save the latest changes when they close an application window. If no button is selected to close the window, NO is returned to the variable COMMAND.

```
commandlist=makelist();
commandlist=insertc(commandlist,
   'You have not saved the latest changes.',1);
commandlist=insertc(commandlist,
   'Do you want to save your changes?',2);
...more SCL statements...
command=messagebox(commandlist,'!','YN','','N','');
commandlist=dellist(commandlist);
```

METHOD

Executes a method block that is defined in an SCL entry

Category: Modular Programming and Object Oriented

Syntax

CALL METHOD(*entry*,*label*<,*parameters*>);

return-value=**METHOD**(*entry*,*label*<,*parameters*>);

entry
is a catalog entry of type SCL. To specify an entry in the current catalog, use *entry* or *entry.type*. To specify an entry in a different catalog, use *libref.catalog.entry.type*. If *type* is not specified, it defaults to SCL.
Type: Character

label
is the name of the method block in the SCL entry.
Type: Character

parameters
are parameters to pass to the method block. The SCL entry that receives these parameters must declare each of them in a METHOD statement.

Note: These parameters are update parameters. See "Input, Output, and Update Parameters" on page 34 for more information. △
Type: Character

return-value
contains the value that is returned by the method block.
Type: Numeric, Character, List, Object, Class, or Interface

Details METHOD can pass parameter values to the called method, and it can receive a value when it is used as a function. In order to return a value, the associated METHOD statement must contain the RETURN= option, and the RETURN statement must specify the variable or literal value to return.

Parameters that are passed must agree with the number of arguments, relative positions, and data types in the corresponding method block unless the REST= or ARGLIST= options are used in the method block. The parameter names in METHOD do not have to match the argument names in the method block.

A method block, which contains a sequence of SCL statements, can be defined either in the current SCL entry or in another, external SCL entry. If the method block is defined in the current entry, it is more efficient to use a LINK statement instead of a METHOD routine.

Parameters are passed in the following ways:

call-by-reference
: passes variables and enables values to be returned to CALL METHOD. This approach enables the called method block to modify values and then to return them.

  An example of a call-by-reference is

```
call method('b.scl','abc',var1,name,field2);
```

  If you do not want to return the values, use the NOCHANGE() routine in the method block. Or, you can assign the INPUT, OUTPUT, and UPDATE options to the variables listed in the METHOD statement to determine which variables can receive and return values. For example:

```
abc: method var1 :input :num
            name :update :char
            field1 :output :num;
```

call-by-value
: is used for all numeric constants, character constants, and expressions. It does not return values to the calling METHOD routine. An example of a call-by-value is

```
call method('b.scl','abc',100,'hello',x+y);
```

Example

Call the method block that is labeled ABC in the SCL entry CODE. The following three parameters are passed: the contents of the variable A, the literal value 3, and the contents of the variable C.

```
call method('code.scl','abc',a,3,c);
```

The method block can return modified values to the variables A and C unless the NOCHANGE routine is specified in the method block or unless A and C are not specified in the METHOD statement as input parameters.

See Also

"DISPLAY" on page 350

"NOCHANGE" on page 569

"METHOD" on page 540

METHOD

Defines a method that can be called by the METHOD routine

Category: Modular Programming and Object Oriented

Syntax

method-name-label:< *method-access-scope* > **METHOD** <*argument-list*>
<OPTIONAL=*argument-list*> <ARGLIST=*arg-list-id* |
REST=*rest-list-id*><RETURN=*data-type*>;

method-name-label

specifies the method name label, which can be up to 32 characters in length. Method labels have the same syntax as SCL labels.

CAUTION:

Leading underscores in method names typically identify methods that are supplied with the SAS System. It is recommended that you do not define method names that have leading underscores unless the underscores are required. For example, you may need to create a new component that supports an interface, such as the staticStringList interface, that has methods that are defined with leading underscores. △

method-access-scope

species how the method can be accessed. If the *method-access-scope* is not provided, a method has **PUBLIC** scope. *Method-access-scope* is valid only for METHOD statements in a CLASS or USECLASS block.

PUBLIC

specifies that the method can be accessed by any SCL program.

PRIVATE

specifies that the method can be accessed only by methods in the same class in which the method is defined. Private methods are not inherited by subclasses of the class.

PROTECTED

specifies that the method can be accessed only by subclasses of the class in which the method is defined. Because a class can be considered a subclass of itself, a protected method can also be accessed from the class in which it is defined.

argument-list

list one or more sets of arguments, with each set specified as follows:

var-list**:<INPUT|UPDATE|OUTPUT>:*****data-type*****(*****length*****)**

var-list

lists one or more variables to contain values that are passed in from a method call using either dot notation or the METHOD, SEND, SUPER, APPLY or SUPAPPLY routine or function. Variables can also be a reference array. Reference array's dimensions are specified by '*'. Comma delimiters are required to separate '*' for multiple dimensions. The actual size of the reference array will be determined at run-time based on the dimensions specified in the array parameter of the calling method. For more information, see "ARRAY" on page 254 and Example 3 on page 546.

INPUT | **I**

specifies that, at run-time, the variable contains the value that is copied from the corresponding parameter of the calling method. However, when the program finishes, the value is not copied back to the calling method. This is equivalent to using CALL NOCHANGE() inside the METHOD block.

UPDATE | **U**

specifies that, at run-time, the variable contains the value that is copied from the corresponding parameter of the calling method. When the program finishes, the value is copied back to that parameter. All the Version 6 SCL method parameters are UPDATE parameter.

OUTPUT | **O**

specifies that, when the program finishes, the value is copied back to the corresponding parameter in the calling program (unless CALL NOCHANGE is specified). An error condition results if the corresponding parameter in the calling program is a constant, because a constant cannot receive a value.

data-type
specifies the type of data that the variable will contain. A named data type (for example, CHAR or LIST) must be preceded by the **:** delimiter. The delimiter is optional for unnamed data types (for example, $).

CHAR<**(***length***)**>
specifies that the variable will contain character data. *Length* can be 1 to 32,767 characters. If *length* is not provided, the default length is 200.

Note: You cannot specify *length* for the **CHAR** *data-type* within the RETURN option. △

LIST
specifies that the variable will contain an SCL list identifier.

NUM
specifies that the variable will contain a numeric value.

OBJECT
specifies that the variable will contain the identifier for an object when it is defined at run time.

This type causes the SCL compiler to generate extra conversion instructions. Consequently, you should use it only when necessary so as to achieve optimal run-time performance.

class-name
specifies that the variable will contain the identifier for an object of the class specified in *class-name*. This type of object is defined at compile time. *Class-name* must be a three- or four-level name unless an IMPORT statement has specified the libref and catalog. In that case, the name can be a one- to four-level name. If the entry type is not specified, it is assumed to be CLASS.

interface-name
specifies that the variable will contain the identifier for an object of the class that supports the interface specified in *interface-name*. *Interface-name* must be a three- or four-level name unless an IMPORT statement has been used to specify the libref and catalog. In that case, the name can be a one- to four-level name.

If the entry type is not specified and a class with that name does not exist, the default entry type of INTERFACE is assumed.

To be compatible with the applications built in earlier releases of SAS software, the **:** delimiter is optional for variables that are declared with unnamed data types (for example, $), but it is required for variables that are assigned named data types. The following example shows a variety of data type declarations including the reference arrays using * as the dimensions:

```
mymethod: method
   char1  : Char(20)
   char2  : Char(10)
   char3  :input :char(50)
   charArr(*):u:char /* a reference array */
   num1   : Num
   num2   : Num
   num3   : num
   numArr(*):num     /* a reference array */
   myList :list
   myObj  :object
```

```
myCol :sashelp.fsp.Collection.class ;
```

Type: Character

length
: is a numeric constant that specifies the length of the preceding variable or variables. The length of a character variable does not have to match the length of the corresponding passed parameter. SCL pads or truncates as necessary. When a length is specified for a variable that is declared as CHAR, the length specification must be enclosed in parentheses.
Type: Character

arg-list-id
: contains the identifier for the SCL list that will contain all the arguments passed to the method. This includes all optional arguments.
Type: List

rest-list-id
: contains the identifier for the SCL list that will contain all arguments that are passed to the method but are not explicitly specified in *argument-list* for either METHOD or OPTIONAL=.
Type: List

data-type
: specifies the type of data that the method can return. The valid data types are Num, Char, List , Object , Array, and Class types.

Details

The METHOD statement enables you to create method blocks and methods for SAS/AF classes. A method block is a feature for defining methods or for making a frequently used routine available to other programs. Methods define the actions for a class. A method block starts with the METHOD labels and ends with an ENDMETHOD statement. Only SCL entries can contain method blocks. Each method block contains additional SCL statements.

RETURN=*data-type* enables you to return a value to the calling method. An error condition is produced if *data-type* is not the same as the type of data to be returned to the calling program. Use a RETURN statement in the method to specify the value to return.

In SCL CLASS statement block or USECLASS statement block, each METHOD statement starts a new local variable scope just like an SCL DO/END block. Parameters with the same name but with the different types can be used across different method statements.

The METHOD statement receives parameters from the calling routine. When there are no optional arguments in the METHOD statement, a strict correspondence is required between the parameters that are passed by the calling routine and the arguments for the METHOD statement. The arguments and parameters must agree in number, data type, and relative position. If the calling program passes an incorrect number of parameters or a parameter of an incorrect type, SCL stops executing the program. The argument-parameter correspondence is less restrictive when you use the options OPTIONAL=, ARGLIST=, and REST= in the METHOD statement:

OPTIONAL=
: enables you to specify one or more optional arguments that are used only if the calling program supplies the corresponding parameters in the parameter list of the calling routine. If corresponding parameters are not supplied, then the optional arguments are initialized to missing values.

ARGLIST= and REST=
: enable you to pass a variable number of parameters to the METHOD statement. You determine the types and order of the variable arguments. The lists identified by *arg-list-id* and *rest-list-id* are created automatically when the entry is called, and they are deleted automatically when the entry ends. When an array is passed as a parameter, the array is expanded into individual items and these items are inserted into the *arg-list-id* and *rest-list-id* lists. ARGLIST= and REST= are mutually exclusive, so you can use only one or the other.

Calling and Executing Method Blocks Other SCL programs call a method block by specifying its label in a dot notation statement or in a METHOD, APPLY, SUPER, SUPAPPLY, or SEND routine or function. Execution of the method block starts at the METHOD statement and ends with the ENDMETHOD statement. After a method block is executed, control returns either to the calling program statement or to the command line. A method block can be tested individually by invoking a TESTAF command with the label=*method-name* option with the SCL debugger. For example, the following statement tests the COMBINE method:

```
testaf label=combine
```

Scope of Method Block Variables All variables that are declared using the DECLARE statement in a method block are local to that method. You cannot use a GOTO statement to jump into a method block in the current entry. All the method parameters are also local to that method if method blocks are written inside a CLASS statement block or a USECLASS statement block.

Passing Parameters to Method Blocks The METHOD statement can receive parameter values for variables that are declared as UPDATE or INPUT. By default, all parameters declared in a METHOD statement are UPDATE parameters.

The parameter-receiving mechanism for the METHOD statement is very similar to that mechanism for the ENTRY statement. The METHOD statement receives parameters from the third argument of the calling METHOD routine. The calling METHOD routine must agree with the corresponding METHOD statement in the following ways (unless OPTIONAL=, ARGLIST=, or REST= are specified):

- The number of arguments received must be the same as the number of parameters passed.
- The relative positions of the arguments passed must match the parameters in the corresponding METHOD statement.
- The data types of both sets of variables must agree.

Otherwise, SCL stops executing the calling METHOD routine and prints an error message.

Returning Modified Parameters to the Calling Routine The METHOD statement can return values to parameters from variables that are declared as UPDATE or OUTPUT. A called method block can modify any argument it receives. However, it cannot return new values to calling routine parameters that are numeric literals, character literals, or expressions. By default, values for variables are returned to the calling routine. If you want a called method block to receive values but not to return values to its calling routine, declare the variables as INPUT. If you want variables in the method to only return values, then declare the method's variables as OUTPUT.

For more information, see "What Happens When Attribute Values Are Set or Queried" on page 122.

Returning a Value to the Calling Routine A METHOD statement can return a value to the calling routine when the METHOD statement uses the RETURN= option to declare the data type of the returned value. A RETURN statement in the method specifies either the variable or expression that contains the value or the literal value to return.

Examples

Example 1: METHOD Statement Declarations Method M1 contains a variety of argument specifications.

```
IMPORT work.myclass.collection.class;
Class Example1;
M1: PUBLIC  METHOD
       /* usenum is UPDATE (default) numeric    */
    usenum :NUM
       /* usechar is UPDATE (default) character */
    usechar :CHAR
       /* mylist is UPDATE (default) list       */
    mylist :LIST
       /* myobject is UPDATE (default) object   */
    myobject :OBJECT
       /* mycoll is UPDATE (default) collection */
    mycoll :COLLECTION
       /* innum is INPUT  numeric               */
    innum :INPUT :NUM
       /* state is OUTPUT character             */
    state :OUTPUT :CHAR
       /* namelist is UPDATE list               */
    namelist :UPDATE :LIST
       /* outputobj is OUTPUT object            */
    outputobj :OUTPUT :OBJECT
       /* amountin is INPUT  numeric            */
    amountin :I :NUM
       /* c3 is OUTPUT character                */
    c3 :O :CHAR
       /* l3 is UPDATE list                     */
    l3 :U :LIST
       /* numarr is a numeric UPDATE array      */
    numarr(5) : NUM
       /* states is a character reference array */
    states(*) : CHAR
       /* return a numeric value                */
    RETURN=NUM;

    ...SCL statements that define the method...
    RETURN(0);

ENDMETHOD;
EndClass;
```

Example 2: Using the RETURN= Option Define an ADD method to add the numbers stored in N1 and N2, and return the sum in the variable TOTAL:

```
Class Example2;
add: public method n1:num n2:num return=num;
   total=n1+n2;
   return(total);
```

```
endmethod;
EndClass;
```

Example 3: Reference Array Whose Size Is Determined at Run Time The following Sort class contains two overloaded methods that are named SORT. Each method contains an array parameter that is a reference array. The size of the reference array will be determined at run time based on the associated array parameters in the calling methods.

```
Class Sort;
/* Generic sort routine for any size of */
/* 1-dimensional numeric array            */
sort: method narr(*):Num;
       /* Find dimensions from the calling program */
   DCL Num temp;
   DCL Num size = dim(narr);
     /* --- Bubble Sort --- */
   do i = 1 to size - 1;
      do j = i+1 to size;
         if narr(i) > narr(j) then
            do;
              temp = narr(i);
              narr(i) = narr(j);
              narr(j) = temp;
            end;
      end;
   end;
   /* Array narr is now sorted in ascending order */
endmethod;

    /* Generic sort routine for any size of */
    /* 1-dimensional  character array       */
sort: method carr(*):Char;
      /* Find dimensions from the calling program */
   DCL Char tempc;
   DCL Num size = dim(carr);
      /* --- Bubble Sort --- */
   do i = 1 to size - 1;
      do j = i+1 to size;
         if carr(i) > carr(j) then
            do;
               tempc = carr(i);
              carr(i) = carr(j);
              carr(j) = tempc;
            end;
      end;
   end;
   /* Array carr is now sorted in ascending order */
endmethod;
EndClass;
```

Example 4: Calling a Method This example creates a new instance of the Sort class and sends a message to the sort method to sort the order of the existing arrays CARR and NARR.

```
Init:
DCL Char(20) carr(3)=('c','b','a');
DCL Num narr(3)={3, 2, 1};
DCL Sort obj = _NEW_ Sort();
obj.sort(carr);
obj.sort(narr);
put carr= narr=;
```

The output is

```
carr=
carr[1]='a'
carr[2]='b'
carr[3]='c'
narr[1] = 1
narr[2] = 2
narr[3] = 3
```

Example 5: Using the REST= Argument Add a variable number of numbers, and print out the sum. The method ignores any character types that are passed in.

```
Class Varying;

SUMPRINT: method msg:Char REST=rest_list;
   DCL num totsum;
   if rest_list = . then
      do;
         put 'No numbers to add were passed in!';
         return;
      end;
   totsum = 0;
   do i = 1 to listlen( rest_list );
      type = itemtype( rest_list, i );
      if ( type = 'N' ) then
         do;
            valn = getitemn( rest_list, i );
            totsum = totsum + valn;
         end;
   end;

   put msg totsum;
endmethod;
EndClass;
```

Use the following program to invoke the SUMPRINT method:

```
Init:
   DCL Varying obj = _NEW_ Varying();
   obj.SUMPRINT('The total is:', 15, 30, 1);
```

The output of this example is

```
The total is: 46
```

Example 6: Parameter Scope and Method Variable Scope This program shows the parameters of the same name and different types being used across different method statements.

```
Class ReUseName;
   m1: Method n:Num c:Char;
        DCL  Num  localN;
        DCL  Char localC;
       EndMethod;
   m2: Method n:Char c:num;
        DCL Char localN;
        DCL Num  localC;
       EndMethod;
EndClass;
```

See Also

"IMPORT" on page 475

METHOD

Contains the name of the method that is currently executing

Category: System Variable

Details _METHOD_ is a character system variable that is provided automatically by the FRAME entry in SAS/AF. However, the SCL compiler does not automatically create a space for it in the SCL data vector. As a result, you get a warning when you compile a FRAME or SCL entry that uses _METHOD_, because the variable is being referenced at compile time but is not assigned a value until run time. You can safely ignore this warning. If you prefer to prevent the warning message from being generated, use the following assignment statement at the top of your program:

```
_method_=_method_;
```

METHOD is useful when you have one or more methods that share the same section of code but which require a CALL SUPER.

In order to use _METHOD_, you must use the DECLARE or LENGTH statement to declare it as a character variable.

METHOD has a valid value only when a method is executing.

Example

For a window control, you may define the _update and _bupdate methods to execute the same section of code if they perform similar functions:

```
length _method_ $40;
BUPDATE:
UPDATE:
   method;
   ...code for _update and _bupdate methods...
   call super(_self_, _method_);
endmethod;
```

Without _METHOD_, you would not know which method to do a CALL SUPER on, so you would have to code the above as

```
BUPDATE:
   method;
   methodName = '_bupdate';
   link update1;
endmethod;
UPDATE:
   method;
   methodName = '_update';
   link update1;
endmethod;

UPDATE1:
   ...code for _update and _bupdate goes here...

   call super(_self_, methodName);
return;
```

See Also

"_SELF_" on page 643

"SUPER" on page 681

"_VALUE_" on page 704

MLENGTH

Returns the maximum length of a variable

Category: Character

Syntax

length=**MLENGTH**(*var*);

length
contains the maximum length of a variable.
Type: Numeric

var
is the variable whose maximum length you want to determine.
Type: Character

Details MLENGTH is different from LENGTH, which returns the trimmed length. For window variables, MLENGTH returns the length of the variable in the display.

If a numeric variable is passed to MLENGTH, MLENGTH always returns a length of 8 for the variable. For non-window variables, MLENGTH returns the declared length of the variable.

Example

In this example, MLENGTH returns the value 5, which is the declared length of variable S. However, LENGTH returns the value 2, because S contains **ab**.

```
length s $ 5;
s='ab';
l=length(s);
m=mlength(s);
```

See Also

"LENGTH" on page 509

MODIFIED

Reports whether a field or FRAME entry control has been modified

Category: Control or Field

Syntax

rc=**MODIFIED**(*wcol-name*);

rc
contains the return code for the operation:

1	modified
0	not modified

Type: Numeric

wcol-name
is the name of the field or FRAME entry control in the window. This name cannot be an element of an array nor an expression. If *wcol-name* is invalid, the program halts.
Type: Character

Details A field's state changes to modified when a user types any character in the field and presses ENTER or a function key or selects a FRAME entry control.

The field or FRAME entry control cannot be an element of an array. To report this information for an array element, use FIELD instead.

The ERRORON statement causes MODIFIED to return a value of 1.

FRAME entry controls can also use the _isModified method.

Examples

Example 1: Opening an FSEDIT Window Open an FSEDIT window for the SAS table specified in the TBLNAME variable. The FSEDIT function displays the table for interactive editing.

```
if (modified(tblname) and tblname ne ' ' ) then
   call fsedit(tblname);
else
   _msg_='Please enter a valid table name.';
```

Example 2: Invalid Syntax for MODIFIED The following are examples of invalid syntax that will not compile:

```
   /* A literal string is used. */
rc=modified('xyz');
   /* Concatenation of two columns. */
rc=modified(a||b);
   /* An array element is used. */
rc=modified(a{i});
```

See Also

"DISPLAYED" on page 353

"ERROR" on page 375

"ERROROFF" on page 376

"ERRORON" on page 377

"FIELD" on page 394

"OBSINFO" on page 575

MODVAR

Changes the name, label, format, or informat of a column in a SAS table

Category: Variable

Syntax

sysrc=**MODVAR**(*table-id*,*var-name*,*new-name*<,*label* <,*format*<,*informat*>>>);

sysrc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

table-id
is the identifier that was assigned when the table was opened. If *table-id* is invalid, the program halts.
Type: Numeric

var-name
is the column whose attribute or attributes you want to change. The column must already exist in the SAS table.
Type: Character

new-name
is the new name to assign to the column. The value must be a valid SAS name and cannot already exist in the SAS table.
Type: Character

label
is the label to assign to the column.
Type: Character

format
is the format to assign to the column.
Type: Character

informat
is the informat to assign to the column.
Type: Character

Details The table must be opened in UTILITY (V) mode, or the function halts.
If you do not want to change an argument, insert a null string ('')as a placeholder.

Example

Change only the label for the column PHONENUM in the SAS table CUSTOMR:

```
dsid=open('customr','v');
if dsid then
   do;
      rc=modvar(dsid,'phonenum','','Office Phone');
      rc=close(dsid);
   end;
```

See Also

"OPEN" on page 576

"VARFMT" on page 705

"VARINFMT" on page 706

"VARLABEL" on page 706

"VARLEN" on page 707

"VARNAME" on page 712

"VARNUM" on page 713

"VARTYPE" on page 716

MOPEN

Opens a member file in a directory

Category: Directory

Syntax

file-id=**MOPEN**(*directory-id*,*member-name*<*open-mode*
<,*record-length*<,*record-format*>>>);

file-id
contains 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 just as you would use a *file-id* returned by the FOPEN function.
Type: Numeric

directory-id
is the identifier that was returned by DOPEN when the directory was opened. If *directory-id* is invalid, the program halts.
Type: Numeric

member-name
is the name of a file in the directory that is identified by *directory-id*.
Type: Character

open-mode
is the type of access to the file:

'A'	APPEND mode, which allows writing new records after the current end of the file.
'I'	INPUT mode, which allows reading only. (This is the default.)
'O'	OUTPUT mode, which defaults to the OPEN mode that was specified in *host-options* in the FILENAME statement or function. If no host option was specified, then OUTPUT mode allows writing new records at the beginning of the file.
'S'	Sequential input mode, which is used for pipes and other sequential devices such as hardware ports.
'U'	UPDATE mode, which allows both reading and writing.

Type: Character

record-length
is 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.
Type: Numeric

record-format
is the record format of the file:

'B'	Interpret data as binary data.
'D'	Use the default record format.
'E'	Use an editable record format.
'F'	The file contains fixed-length records.
'P'	The file contains printer carriage-control characters in a host-dependent record format.
'V'	The file contains variable-length records.

To use the existing record format, do not specify a value here.
Type: Character

Details

CAUTION:
Use OUTPUT mode with care. Opening an existing file for output may overwrite the current contents of the file without warning. △

The member file 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, using MOPEN saves you from having to use a separate fileref for each member.

If the file already exists and is opened with an *open-mode* of O, then the output mode defaults to either APPEND or REPLACE, based on *host-options* that were specified in the FILENAME function or statement. For example:

```
rc=filename('file',filename,' ','mod');
fid=fopen('file','o');
rc=fput(fid,'This is a test.');
rc=fwrite(fid);
rc=fclose(fid);
```

If FILE already exists, then FWRITE appends a new record. However, if no host option was specified with the FILENAME function, then FWRITE writes the value at the beginning of the file, which could replace an existing value.

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 managed by the host operating system. Different host operating systems identify such groupings with different names, such as directory, subdirectory, MACLIB, or partitioned data set. See the SAS documentation for your operating environment for details.

Opening a directory member for output is not possible on some operating systems. △

Example

Assign the fileref MYDIR to a directory. Then open the directory, determine the number of members, retrieve the name of the first member, and open that member. The last three arguments to MOPEN are the defaults.

```
rc=filename('mydir','filename');
did=dopen('mydir');
frstname=' ';
memcount=dnum(did);
if (memcount>0) then
   do;
      frstname=dread(did,1);
      fid=mopen(did,frstname,'i',0,'d');
      ...SCL statements to process the member...
      rc=fclose(fid);
   end;
else
   _msg_=sysmsg();
rc=dclose(did);
```

See Also

"DOPEN" on page 357

"FCLOSE" on page 386

"FOPEN" on page 420

MSG

Contains the text to display on the window's message line the next time the window is refreshed

Category: System Variable

Details _MSG_ is a system variable that is created for every SCL program you compile. The compiler creates a space for _MSG_ in the SCL data vector.

Typically an application displays error and warning messages on the window's message line. The text for system error and warning messages can be obtained by using the SYSMSG or STDMSG functions. You can also assign your own text to the _MSG_ variable. Messages are displayed when the window is refreshed.

FRAME entries can also use the _getMsg and _setMsg methods to query and update the _MSG_ variable.

On some operating systems, _MSG_ is not displayed if the window has BANNER set to NONE.

Example

Display a message if a table cannot be opened:

```
INIT:
   dsid = open('sasuser.class');
   if dsid eq 0 then
      _msg_ = sysmsg();
return;
```

See Also

"SYSMSG" on page 686

NAMEDITEM

Returns the index of a named item in a list

Category: List

Syntax

index=**NAMEDITEM**(*list-id*,*name*<,*occurrence*
    <,*start-index*<,*forceup*>>>);

index
contains the position of the item in the list, or 0 if the named item is not found.
Type: Numeric

list-id
is the identifier of the list that NAMEDITEM searches. An invalid *list-id* produces an error condition.
Type: Numeric or List

name
is the name of the item to search for. If *name* is specified, then trailing blanks are removed before the search. If *name* is blank, the first unnamed item is returned.
Type: Character

occurrence
specifies which occurrence of the named item to search for. The default, 1, specifies the first occurrence of the item.
Type: Numeric

start-index
specifies where in the list to begin searching for the item. By default, *start-index* is 1 (the first item). If *start-index* is positive, then the search begins at position *start-index* items from the beginning of the list. If *start-index* is negative, then the search begins at the item specified by ABS(*start-index*) items from the end of the list. An error condition results if the absolute value of *start-index* is zero or if it is greater than the number of items in the list.
Type: Numeric

forceup
can have one of the following values:

'Y'	specifies a case-insensitive search, which overrides the HONORCASE or NOHONORCASE list attribute.
'N'	specifies a search that uses the HONORCASE or NOHONORCASE list attribute and is the default action for lists when FORCEUP is not specified.
IGNORECASE	is an alias for NOHONORCASE.

Details NAMEDITEM searches only the top level of the list specified by *list-id*. That is, it does not search sublists. Several functions that access items in a list by position have counterparts that access items by their names such as GETITEMC versus GETNITEMC. Because it is more efficient to retrieve an item by its position rather than by its name, you can use NAMEDITEM to find the position and then use the functions that access items by position rather than by name.

If *occurrence* and *start-index* are both positive or both negative, the search proceeds forward from the *start-index* item. For forward searches, the search continues only to the end of the list and does not wrap back to the front of the list. If *occurrence* or *start-index* is negative, the search is backwards. For backward searches, the search continues only to the beginning of the list and does not wrap back to the end of the list.

Example

Swap the numeric values associated with the first and last occurrence of the item named **x**:

```
/*  Return first occurrence of X.  */
first=getnitemn(listid,'X');
/*  Return last occurrence of X.   */
last=getnitemn(listid,'X',1,-1);
list=setnitemn(listid,last,'X');
list=setnitemn(listid,first,'X',1 -1);
```

The following example shows a slightly more efficient way to perform the swap operation. This method does not require a second search for the item, and it can also detect when item **x** does not exist in the list.

```
/*  Return the position number of the */
/* first item X. */
ifirst=nameditem(listid,'X');
if (ifirst>0) then
   do;
      first=getitemn(listid,ifirst);
/* Return the position of the last item X.*/
      ilast=nameditem(listid,'X',1,-1);
      list=setitemn(listid,getitemn(listid,ilast),
                   ifirst);
      list=setitemn(listid,first,ilast);
   end;
```

Note: This example checks to see whether there is at least one item named **x** but never checks to see whether there is another item named **x**. It assumes that there is at least one more item named **X** △

See Also

"DELNITEM" on page 337

"NAMEITEM" on page 558

"SEARCH" on page 635

NAMEDIVIDE

Returns the number of parts of a compound name as well as the values of each part

Category: Utility

Syntax

rc=**NAMEDIVIDE**(*name*,*num-parts*, *part-1*, . . . *part-4*);

rc

contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

name

is the name of a two- to four-part compound name to be separated.

Type: Character

num-parts

returns the number of parts in *name*.

Note: This parameter is an update parameter. See "Input, Output, and Update Parameters" on page 34 for more information. △

Type: Numeric

part-1. . . part-4

contains the values of the first through fourth parts of *name*.

Note: These parameters are an update parameters. See "Input, Output, and Update Parameters" on page 34 for more information. △

Type: Character

Details NAMEDIVIDE divides a compound name into parts and returns the number of parts as well as the value of each part.

Note: NAMEDIVIDE does not validate individual parts of the name. △

Example

Divide NAME into parts and invoke DIALOG if NAME is a FRAME entry. If NAME is not a FRAME entry, CALL DISPLAY.

```
rc=namedivide(name,parts,part1,part2,part3,part4);
   /* Was NAMEDIVIDE successful? */
if (rc^=0) then do;
   put 'Error: Invalid name ' name;
   return;

type=upcase(part4);
if (part4='FRAME') then
   call dialog(name);
else
   call display(name);
```

See Also

"NAMEMERGE" on page 559

NAMEITEM

Returns and optionally replaces the name of an item in an SCL list

Category: List

Syntax

item-name=**NAMEITEM**(*list-id*<,*index*<,*new-name*>>);

item-name
contains the name of the specified list item, or a blank if the item does not have a name.
Type: Character

list-id
is the identifier of the list that contains the indexed item. An invalid *list-id* produces an error condition.
Type: Numeric or List

index
is the position of the item in the list. The position can be specified as a positive or negative number. By default, *index* is 1 (the first item). If *index* is a positive number, then the item is at position *index* from the beginning of the list. If *index* is a negative number, then the item is at position ABS(*index*) from the end of the list. An error condition results if the absolute value for *index* is zero or if it is greater than the number of items in the list.
Type: Numeric

new-name
is the new name to assign to the list item.
Type: Character

Details NAMEITEM returns the name of the item at the list position specified by *index*. If a value for *new-name* is also provided, then NAMEITEM assigns that name to the item, replacing the old name.

An error condition results if the value for *new-name* is provided and the list has any of the following attributes:

- NOUPDATE
- SASNAMES, and *new-name* is not a valid SAS name
- NODUPNAMES, and *new-name* duplicates the name of another item in the list.

To check the attributes of a list or list item, use HASATTR. To change attributes, use SETLATTR.

See Also

"DELNITEM" on page 337

"NAMEDITEM" on page 555

NAMEMERGE

Returns a compound name by merging name parts

Category: Utility

Syntax

name=**NAMEMERGE**(*part-1*,*part-2*<,*part-3*<,*part–4*>>);

name
contains a two- to four-part compound name. Name is blank if any error is detected.

part-1 . . . part-4
contain the name segments to be merged. Both *part-1* and *part-2* must be specified.

Details NAMEMERGE creates a SAS name by merging the values that are stored as two to four parts. If four parts are specified, then *part-1* and *part-4* can be blank.

Example

Create a CATALOG entry name, using the values stored in the variables LIBREF, CATALOG, NAME, and TYPE. Then run the CATALOG entry through DISPLAY.

```
entry=namemerge(libref,catalog,name,type);
if (entry^=' ') and (cexist(entry)) then
    call display(entry);
```

See Also

"NAMEDIVIDE" on page 557

NEO

Creates an object

Category: Object Oriented

Syntax

object-id=**_NEO_** *class-name*(<*init-arg*<,*new-arg-1*<, . . . ,*new-arg-n*>>>);

object-id
contains the identifier for the new object.
Type: Numeric or Object

class-name
is the name of the class from which to create the object. This can be a one- to four-level name. If *class-name* is a one- or two-level name, and if the CLASS entry that defines *class-name* is not in the application catalog, then *class-name* must exist in one of the catalogs defined by the IMPORT statement. Otherwise, the compiler produces an error message.

init-arg
is the argument to pass to the _init method for the new object.
Type: Character

new-arg
are additional arguments to pass to the _new method of the new object.
Type: Character

Details The _NEO_ operator provides a faster and more direct way to create an object. It combines the actions of loading a class with LOADCLASS and initializing the object with the _new method, which invokes the object's _init method.

Example

Create a frame and enter the following source code:

```
import sashelp.classes;

init:
  dcl list AttrList RegionList;
  dcl checkbox_c c;

  AttrList   = makelist();
  RegionList = makelist();

  startcol=10;  startrow=10;
  rc = setniteml(AttrList, RegionList, '_region_');
  rc = setnitemn(AttrList, -1, 'num');
  rc = setnitemn(RegionList, startcol, 'ulx');
  rc = setnitemn(RegionList, startrow, 'uly');
  rc = setnitemc(RegionList, 'simple', 'border_style');
  rc = setnitemn(RegionList, 5, 'border_width');
  rc = setnitemc(RegionList, 'red', 'border_color');

  c = _neo_ checkbox_c(attrlist);
  return;
```

Note that you cannot use the _NEW_ operator to do this, because _NEW_ passes its arguments to a constructor, whereas _NEO_ passes them to the _new method (which is what you want in this case). Even if you created a constructor for the checkbox, it would not work with the _NEW_ operator because the checkbox will be displayed before the constructor has a chance to run. Therefore, you must either resort to the old behavior using CALL SEND, or use the _NEO_ operator as shown above.

See Also

"DECLARE" on page 330

"IMPORT" on page 475

"INSTANCE" on page 485

"LOADCLASS" on page 525

NEW

Defines a new SAS table interactively

Category: SAS Table

Syntax

CALL NEW(*table-name*,*model-table*,*num-row*, *display-window*);

table-name
is the SAS table to be created, specified as <*libref*.>*member*. The default library, USER, is used if *libref* is omitted. If a null string **('')** is specified for *table-name*, then the DATA*n* naming convention is used to create a table in the USER library.
Type: Character

model-table
is an existing SAS table after which the new SAS table is to be modeled. Use a null string ('') as a placeholder if you do not want to specify a model table.
Type: Character

num-row
is the number of initial rows for the new SAS table. This value must be equal to or greater than 0. If the default is set to **'0'**, then the value is a negative number. All columns in all rows initially contain missing values. The value for *num-row* cannot be a missing value.
Type: Character

display-window
specifies whether the NEW window is displayed so that column definitions for the new SAS table can be edited before the table is created:

'Y'	displays the NEW window to allow editing of the column names and attributes before the new SAS table is created. (This is the default.)
'N'	does not display the NEW window. The column definitions in the new SAS table will be an exact replica of those in the model table. A value must also be supplied for *model-table*.

Type: Character

Details NEW creates a new blank SAS table. However, it does not replace an existing table.

CAUTION:

Specifying an existing SAS table for *table-name* causes a program halt. To prevent the program halt, use EXIST to determine whether the table already exists, DELETE to delete the table, and CALL NEW to create a new table with the same name. △

By default, the routine opens the NEW window to enable a user to interactively define the names and attributes of the columns in the new SAS table. The NEW window that this routine opens is the same as the window that is displayed when the NEW= option is used with the PROC FSEDIT or PROC FSVIEW statement.

You can specify *model-table* so that all the names and attributes of the model are automatically copied to the new table. (Only the column names and column attributes of the model table are copied, not the values that it contains.) When you specify a model table, you can use *display-window* to bypass the NEW window and create the new table with the same column names and attributes as the model table. Open the NEW window only if you want to enable users to alter the column names and attributes before the new table is created.

Use *num-row* to specify how many blank rows to create for the new SAS table. All columns in all rows of the new table initially contain missing values.

Examples

Example 1: Creating a SAS Table from a Model Table Create a new SAS table with a name that the user supplies in the field for the window variable TABLE. Before attempting to create the table, determine whether a SAS table with the specified name already exists. If so, issue a message on the application window's message line. The new table is modeled after the existing table MODEL.A, and the user is not given the opportunity to modify the column definitions (the NEW window is not opened). The new table is created with ten blank rows.

```
if (exist(table)) then
 _msg_='SAS table '||table||' already exists.';
else
   do;
      call new(table,'model.a',10,'n');
      _msg_='SAS table '||table||' has been created.';
   end;
```

Example 2: Creating a SAS Table from a User Definition Create a new SAS table with a name that the user supplies in the field for the window variable TABLE. These statements display the NEW window for the user to define the columns in the table. No model table is used.

```
if (exist(table)) then
  _msg_='SAS table '||table||' already exists.';
else
   do;
      call new(table,' ',1,'y');
      _msg_='SAS table '||table||' has been created.';
   end;
```

See Also

"FSEDIT" on page 433

"FSVIEW" on page 437

"NEWVAR" on page 565

"OPEN" on page 576

NEW

Creates an object and runs an associated class constructor

Category: Object Oriented

Syntax

object-id=**_NEW_** *class-name*(<*arg1, arg2, . . . , argn*>);

object-id
contains the identifier for the new object.
Type: Numeric, Classes, and Object

class-name
is the name of the class from which to create the object. This can be a one- to four-level name. If *class-name* is a one- or two-level name, and if the CLASS entry that defines *class-name* is not in the application catalog, then *class-name* must exist in one of the catalogs defined by the IMPORT statement. Otherwise, the compiler produces an error message.

arg1, arg2, . . . , argn
are the arguments that are passed to the class constructor.
Type: Character

Examples

Example 1: Creating an Instance of an Object and Running a Class Constructor The _NEW_ operator enables you to create an instance of an object and run a class constructor all in one step.

```
X.SCL
class Complex;
  private num x y;

  Complex: method a1: num a2: num;
           x = c.x;
           y = c.y;
  endmethod;
endclass;

Y.SCL
init:
  dcl Complex c = _new_ Complex(1,2);
```

```
    dcl Complex c2 = _new_ Complex(c);
    return;
```

Example 2: Using a Class That Has an Overloaded Constructor Where you have a class, Complex, that has an overloaded constructor, one version takes two numeric arguments, and the other takes a complex number.

In the first _NEW_ statement, you create a complex number, **`1 + 2i`**, by passing in the real and imaginary parts (1 and 2) as arguments to the _NEW_ operator. _NEW_ first creates an instance of the Complex class, and then calls the appropriate constructor based on its argument list. In this case, it calls the constructor that takes two numeric arguments.

In the second _NEW_ statement, you create a complex number by passing in another complex number (in this case, the first one you created, **`1 + 2i`**). _NEW_ calls the second constructor in the Complex class — the one that takes a complex number as an argument.

Constructors must always have the same name as the class in which they are located, and they must always be void methods (that is, they do not return a value). Constructors provide a convenient way to initialize a class before you begin using it.

For example, if you didn't use constructors in the above example, you would have to create another set of methods to initialize the complex number and call those separately, as in the following example:

```
  X.SCL
  class Complex;
    private num x y;

    set: method a1: num a2: num;
             x = a1;
             y = a2;
    endmethod;

    set: method c: Complex;
             x = c.x;
             y = c.y;
    endmethod;
  endclass;

  Y.SCl
  init:
   dcl Complex c = _new_ Complex();
   dcl Complex c2 = _new_ Complex();
   c.set(1,2);
   c2.set(c);
   return;
```

You can overload constructors as shown above, or you can have only one constructor, or you can have none. If you don't supply a constructor in a class, the _NEW_ operator still attempts to run a parent class constructor. In the following example, the _NEW_ operator for the class Y calls the constructor in X, since X is the parent of Y.

```
  X.SCL
  class x;
   x: method n: num;
      put n=;
   endmethod;
```

```
endclass;

Y.SCL
class y extends x;
endclass;

Y.SCL
init:
 dcl y y = _new_ y(100);
 return;
```

This behavior applies to all classes that are instantiated by using _NEW_ — even those that have no explicit constructors.

If you do not supply a constructor for your classes, you can still use _NEW_ with no arguments because OBJECT.CLASS (which all classes extend) contains a constructor that takes no arguments and performs no actions. This "dummy" constructor needs to be overridden if you want to supply a constructor that does not take an argument by using the following:

```
class x;
 x: method /(state='0');
    put 'in x constructor';
 endmethod;
endclass;
```

Constructors can be used only in conjunction with the _NEW_ operator. They are not called if the object is instantiated in some other way, such as by using the _NEW_ method in CALL SEND, or by dropping the object in a frame. In particular, in the case of a visual object, you cannot run a constructor after the object is instantiated, but only before it is displayed. Therefore, you cannot use a constructor to initialize the object. For these types of objects, you must either use the _NEO_ operator or use the _NEW_ method with CALL SEND.

See Also

"DECLARE" on page 330

"IMPORT" on page 475

"LOADCLASS" on page 525

"_NEO_" on page 559

NEWVAR

Adds a column to a new SAS table

Category: SAS Table

Syntax

sysrc=**NEWVAR**(*table-id*,*col-name*,*type*<,*length*<,*label*
<,*format*<,*informat*>>>>);

sysrc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

table-id
is the identifier that was assigned when the table was opened. If *table-id* is invalid, the program halts. The SAS table must be opened with an open mode of NEW.
Type: Numeric

col-name
is the name to assign to the new column. The name must be a valid SAS name.
Type: Character

type
specifies the data type of the new column:

'C'	character
'N'	numeric

Type: Character

length
is the length of the new column. For a character column, this value can be between 1 and 32,767. For a numeric column, it can be between 3 and 8. On some systems, the minimum length can be 2.
Type: Numeric

label
is the label for the new column. This is a character string from 1 to 40 characters. If the label contains single quotes, use two single quotes for each of these internal single quotes, or surround the label with double quotes. Each set of two single quotes counts as one character.
Type: Character

format
is the format for the new column. This must be a valid SAS format name. Formats can be either defined by the user or supplied by SAS software. If not specified, SAS assigns a default format. If *type* is **C**, then a character format should be specified. Otherwise, a numeric format should be specified.
Type: Character

informat
is the informat for the new column. This must be a valid SAS informat name. Informats can be either defined by the user or supplied by SAS software. If not specified, SAS assigns a default informat. If *type* is **C**, then a character informat should be specified. Otherwise, a numeric informat should be specified.
Type: Character

Details

CAUTION:
The table must be opened in NEW mode, or SCL halts the program. Opening an existing SAS table in NEW mode creates a new table that overwrites the old table. When the table is closed after using NEWVAR, the SAS table is created with zero rows. If the column name matches a column name that was defined for the table by a previous NEWVAR function, the new definition overrides the previous definition. △

Example

Open a new SAS table, MYDATA, in NEW mode, and then add the column AGE, which is numeric, and NAME, which is character. AGE has a length of 8, and NAME has a length of 20. The label of NAME is set to **`Last Name`**.

```
dsid=open('mydata','n');
rc=newvar(dsid,'num','n',8);
rc=newvar(dsid,'char','c',20,'Last Name');
rc=close(dsid);
```

See Also

"OPEN" on page 576

NEXTCMD

Discards the current command on the command line

Category: Command

Syntax

CALL NEXTCMD();

Details NEXTCMD deletes the words up to the next semicolon or up to the end of the command. If a semicolon is not found, the entire contents of the command line are deleted.

Ordinarily, you clear the command line after reading a command or a series of commands. NEXTCMD is usually used in conjunction with CONTROL ENTER, ALWAYS, or ALLCMDS.

If the command line contains two or more commands separated by semicolons, then only the first command on the command line is executed during the current execution of the MAIN section. The next command is executed when control is returned to the program or when another entry is displayed, as in the use of the DISPLAY routine.

Example

Suppose you have an FSEDIT application and you want to prevent everyone but one user from deleting rows. You can use NEXTCMD to remove all commands that start with DEL for users other than USERAA.

```
FSEINIT:
    control always;
return;

INIT:
return;

MAIN:
if word(1,'u') =: 'DEL' and
```

```
symget('sysjobid') ne 'USERAA' then
   do;
      call nextcmd();
      _msg_=
      'You are not authorized to delete rows';
   end;
return;

TERM:
return;
```

Issue the command **AXX BXX; CXX DXX**. After NEXTCMD is called, only **CXX DXX** remains in the command buffer. After NEXTCMD is called a second time, no commands remain to be processed.

See Also

"NEXTWORD" on page 568

"WORD" on page 725

NEXTWORD

Deletes the current word and advances to the next word in the current command

Category: Command

Syntax

CALL NEXTWORD();

Details

A *word* is the text at the current position up to the next blank or semicolon. A semicolon denotes the end of a command in addition to the end of a word. NEXTWORD is used with WORD and is usually used in conjunction with CONTROL ENTER, CONTROL ALWAYS, or CONTROL ALLCMDS.

If the command line contains two or more commands separated by semicolons, then only the first command on the command line is executed during the current execution of the MAIN section. The next command is executed when control is returned to the program or when another entry is displayed, as in the use of the DISPLAY routine.

Example

If a user issues the command **AXX BXX CXX DXX**, the succession of words is as follows:

	Initial	**After 1st NEXTWORD**	**After 2nd NEXTWORD**
word(1)	AXX	BXX	CXX
word(2)	BXX	CXX	DXX
word(3)	CXX	DXX	*blank*

See Also

"NEXTCMD" on page 567

"WORD" on page 725

"WORDTYPE" on page 726

NOCHANGE

Causes the called program to return the original values for the variables that it received as parameters in an ENTRY statement

Category: Modular Programming and Object Oriented

Syntax

CALL NOCHANGE();

Details The NOCHANGE routine causes a called program to return the original values for the variables that it received as parameters in an ENTRY statement, disregarding any changes that subsequently may have been made to those variables. This routine is used in the called program that contains the ENTRY statement.

NOCHANGE has no effect on reference arrays. See "ARRAY" on page 254 for more information about reference arrays. NOCHANGE overrides the I/O/U mode that was specified for parameters in the ENTRY and METHOD statements.

Example

Suppose that A.SCL calls B.SCL and passes to it the parameters X, Y, and Z. B.SCL tests the _STATUS_ variable for the value C (indicating that a CANCEL command has been issued). If a user exits B.SCL with a CANCEL command, then no updated values are returned to A.SCL.

Here is the A.SCL program:

```
INIT:
return;

MAIN:
call display('mylib.test.b.scl',x,y,z);
return;

TERM:
```

```
return;
```

Here is the B.SCL program:

```
entry a b c 8;
INIT:
return;

MAIN:
   ...SCL statements...
return;

TERM:
   if _STATUS_='C' then call nochange();
return;
```

See Also

"APPLY" on page 252

"DISPLAY" on page 350

"ENTRY" on page 369

"METHOD" on page 540

"NOTIFY" on page 572

"SEND" on page 644

"SUPAPPLY" on page 679

"SUPER" on page 681

NOTE

Returns an identifier for the current row of a SAS table

Category: SAS Table

Syntax

note-id=**NOTE**(*table-id*);

note-id
contains the identifier that is assigned to the row.
Type: Numeric

table-id
is the identifier that was assigned when the table was opened. If *table-id* is invalid, the program halts.
Type: Numeric

Details You can use *note-id* in the POINT function to return to the current row. (Use NOTE to mark a row, then use POINT to return to the row.) Each *note-id* is a unique numeric value. There can be up to 1,000 *note-ids* per open table.

To free the memory that is associated with a row ID, use DROPNOTE.

Example

Use NOTE to return the identifier for the 10th row. Then use POINT to point to the row that corresponds to NOTEID.

```
INIT:
   tableid=open('sasuser.fitness','i');
return;
MAIN:
   /* Read row 10 */
rc=fetchobs(tableid,10);
if (abs(rc) ne 0) then
    do;
       put "Read operation failed";
       return;
    end;
   /* Display the row number in the LOG window */
cur=curobs(tableid);
put "CUROBS=" cur;
   /* Mark row 10 */
noteid=note(tableid);
   /* Rewind the pointer to the beginning of the table */
rc=rewind(tableid);
   /* Read first row */
rc=fetch(tableid);
   /* Display the row number */
cur=curobs(tableid);
put "CUROBS=" cur;
   /* POINT to row 10 marked earlier by NOTE */
rc=point(tableid,noteid);
   /* Read the row */
rc=fetch(tableid);
   /* Display the row number to confirm it is 10 */
cur=curobs(tableid);
put "CUROBS=" cur;
return;

TERM:
   if (tableid >0) then rc=close(tableid);
return;
```

The output produced by this program is

```
CUROBS=10
CUROBS=1
CUROBS=10
```

See Also

"DROPNOTE" on page 361

"POINT" on page 592

"REWIND" on page 618

NOTIFY

Sends a method to a control that is identified by its name

Category: Object Oriented

Valid: FRAME programs only

Syntax

CALL NOTIFY(*control-name,method-name*<*,parameters*>);

control-name
is the control to send the method to, or `'.'` (a string containing a period) to send the method to the FRAME entry component.
Type: Character

method-name
is the method to invoke.
Type: Character

parameters
specify additional numeric or character arguments, separated by commas, that are required by the method.

Note: These parameters are update parameters. See "Input, Output, and Update Parameters" on page 34 for more information. △
Type: Numeric, Character

Details NOTIFY sends a method to a control in a FRAME entry by specifying the control's name. NOTIFY may be called only from the SCL program for the FRAME entry to which the control belongs, because that is the only code in which the control's name is not ambiguous. You can also use NOTIFY as a function if the called method returns a value with a RETURN statement in the program that defines the method.

Note: As with DISPLAY, the SCL compiler cannot determine which data types are expected for each of the parameters passed to a method. When the application executes, SAS/AF software verifies that each parameter is correct. If there is a type error, the SCL program halts. △

In most cases, put quotation marks around the control name so that the value of the control is not passed to NOTIFY. For example, the following code hides the Version 6 control named BUTTON:

```
CALL NOTIFY('button','_hide');
```

However, if BUTTON has a value of **OK**, the following code hides the control named OK:

```
CALL NOTIFY(button,'_hide');
```

If the value of BUTTON is not the name of a control, the program halts.

Similarly, be sure to put quotation marks around *method-name* unless *method-name* is an expression.

For more information about dot notation, see “Accessing Object Attributes and Methods With Dot Notation” on page 119.

Example

Suppose you have a FRAME entry with two Version 6 pushbuttons, **OK** and **NOT_OK**. The following code causes **OK** to be grayed when a user clicks on **NOT_OK**:

```
NOT_OK:
   call notify('ok','_gray');
return;
```

The following example fills a list with the set of controls in the FRAME entry:

```
controls=makelist();
call notify('.','_getWidgets',controls);
```

See Also

“APPLY” on page 252

“INSTANCE” on page 485

“LOADCLASS” on page 525

“LOADRES” on page 526

“SEND” on page 644

“SUPAPPLY” on page 679

“SUPER” on page 681

NSELECT

Returns the number of rows that have been selected in a selection list

Category: Extended Table

Syntax

num-rows=**NSELECT**();

num-rows

contains the number of selected rows.

Type: Numeric

Details Because you can define extended tables only in SAS/AF software, you cannot use NSELECT in FSEDIT or FSVIEW programs. NSELECT is valid only for PROGRAM entries. FRAME entry controls must use the _getNselect method.

In order for an extended table to be considered a selection list, you must specify a number of selections in the SETROW routine.

Example

Return the number of selected rows:

```
nsel = nselect();
_msg_ ='You have selected ' || nsel || ' rows.';
```

See Also

"ISSEL" on page 495

"SELECT" on page 639

"SELECTED" on page 642

"UNSELECT" on page 696

NUMFKEYS

Returns the number of function keys that are available for the device

Category: Keys

Syntax

fkeynum=**NUMFKEYS**();

fkeynum
contains the number of function keys that are available for the device.
Type: Numeric

Details You can use this function only in entries that have a window variable or text in the DISPLAY window.

You can use NUMFKEYS when you want to use SCL to return the number of function keys, then disable procedure-specific commands and SAS windowing commands and redefine the function keys while a window is open. When you redefine the function keys, you can limit the commands that can be used in an application window. (If you do this, you may want to restore the settings with SETFKEY before the application window closes.)

Example

Assign a custom PMENU. Then use NUMFKEYS to find out how many function keys a user's device has. Use SETFKEY first to disable them and then again to restore the settings when the window closes.

```
dcl char command(30);

INIT:
/* Assign the PMENU entry to the window.  */
   rc=pmenu('editdata.pmenu');
/* Turn the PMENU facility on.  */
   call execcmd('pmenu on');
```

```
/* Execute the MAIN section even if a user makes */
/* an error or issues an unknown command.  */
   control enter;
/* Determine the number of function keys */
/* on a user's keyboard.  */
   numkeys=numfkeys();
   do n=1 to numkeys;
      command{n}=getfkey (fkeyname(n));
  /* Disable function key assignments.  */
      call setfkey(fkeyname(n),'');
   end;
return;
MAIN:
    ...Statements to process custom commands...
return;
TERM:
/* Restore command assignments to function keys.*/
   do n=1 to numkeys;
      call setfkey(fkeyname(n),command{n});
   end;
/* Turn the PMENU facility off.  */
   call execcmd('pmenu off');
return;
```

See Also

"FKEYNAME" on page 413

"GETFKEY" on page 440

"SETFKEY" on page 649

OBSINFO

Returns information about the current row in an FSEDIT application

Category: SAS Table

Valid: FSEDIT programs only

Syntax

rc=**OBSINFO**(*info-item*);

rc
contains the return code for the operation.
Type: Numeric

info-item
is one information item from the following list:

'ALTER'
reports whether the currently displayed row can be edited:

0	The row cannot be edited. If ALTER=0 and LOCKED=0, the row is open in FSBROWSE. If ALTER=0 and LOCKED=1, the row is open in FSEDIT, but the row is locked.

1	The row can be edited. If ALTER=1 and LOCKED=0, then the row is open in FSEDIT.

'CUROBS'
reports the number of the current row:

n	The number of the row.
–1	The table is accessed using an engine that does not support row numbers.

CUROBS returns a missing value (.) when there are no rows in the table, when no rows meet the specified WHERE condition, and when you are on a deleted row.

'DELETED'
reports whether the currently displayed row is marked for deletion:

1	The row is marked for deletion.
0	The row is not marked for deletion.

'LOCKED'
reports whether the currently displayed row is locked by another user:

1	The row is locked.
0	The row is not locked.

'MODIFIED'
reports whether a value has been changed in any table column in the currently displayed row or the row is new:

1	Either column has been changed or the row is new.
0	No table columns have been changed.

'NEW'
Reports whether the currently displayed row is a new row:

1	The row is new.
0	The row already exists in the table.

Type: Character

Example

Return information for the row that is currently displayed in the FSEDIT window:

```
rc=obsinfo('curobs');
rc=obsinfo('deleted');
rc=obsinfo('locked');
rc=obsinfo('new');
rc=obsinfo('modified');
```

See Also

"CUROBS" on page 321

OPEN

Opens a SAS table

Category: SAS Table

Syntax

table-id=**OPEN**(<*table-name*< >,*mode*>);

table-id

contains the table identifier, or 0 if the table could not be opened.

Type: Numeric

table-name

is the SAS table or SAS/ACCESS view descriptor to open, specified as <*libref*.>*member-name*<(*data-set-options*)>. The default value for *table-name* is _LAST_, which is the last table created in the current SAS session. A one-level name is assumed to be USER.*member-name*.

All SAS data set options are valid except the FIRSTOBS= and OBS= options, which are ignored.

Type: Character

mode

specifies the type of access to the table:

'I'	an INPUT mode in which values can be read but cannot be modified. (This is the default.) Rows are read in random order.
'IN'	an INPUT mode in which rows can be read sequentially and can also be revisited.
'IS'	an INPUT mode in which rows can be read sequentially but cannot be revisited.
'N'	NEW mode, which creates a new table. If *table-name* already exists, the table is replaced without warning.
'U'	an UPDATE mode in which values in the table can be modified and rows can be read in random order.
'UN'	an UPDATE mode in which values in the table can be modified, rows can be read sequentially, and rows can be revisited.
'US'	an UPDATE mode in which values in the table can be modified and rows can be read sequentially. However, rows cannot be revisited.
'V'	UTILITY mode, which must be used in order to change any column attributes or to manipulate any associated table indexes.

Type: Character

Details OPEN opens a SAS table or a SAS/ACCESS view descriptor and returns a unique numeric table identifier, which is used in most other SCL functions that manipulate tables.

If *mode* is I or U , then OPEN defaults to 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 or UN mode automatically. Files are opened with sequential access, and a system level warning is set. For example, opening a DB2 SAS/ACCESS view descriptor in INPUT (I) mode opens the file but produces the warning "This task requires reading rows in a random order, but the engine allows only sequential access." To enable the file to be read multiple times and to prevent this warning, use an open mode of IN instead.

Note that both IS and IN (as well as US and UN)refer to sequential access. However, IN allows revisiting a row, whereas IS does not.

Note: If sequential access is too restrictive but random access is too slow, try specifying the TOBSONO= data set option. See *SAS Language Reference: Dictionary* for more information. △

By default, a SAS table is opened with a control level of RECORD. See *SAS Language Reference: Dictionary* for details about the CNTLLEV (control level) SAS data set option.

A table that is already opened can be opened again, subject to the following restrictions:

- If the table is already opened in UPDATE or INPUT mode, it cannot be opened again in UTILITY mode.
- If the table is already opened in UTILITY mode (so that columns can be dropped, inserted, or changed), it can only be opened again in NEW mode.

A table that is already open in any mode can be opened again in NEW mode, because that replaces everything in the old table.

An open SAS table should be closed when it is no longer needed.

Example

Open the table PRICES in the library MASTER, using INPUT mode:

```
tableid=open('master.prices','i');
if (tableid=0) then _msg_=sysmsg();
else _msg_='PRICES table has been opened';
```

You can pass values from SCL variables to be used in data set options. Open the table MYDATA, and use the WHERE= data set option to apply a permanent WHERE clause, using the value from the numeric variable SCRNUM:

```
tableid=
open('mydata(where=(num='||put(scrnum,5.)||'))');
```

Open the table MYDATA, and use the WHERE= data set option to apply a permanent WHERE clause, using the value from the character variable SCRNAME:

```
tableid=
open('mydata(where=(name='||quote(scrname)||'))');
```

See Also

"CLOSE" on page 292

"MODVAR" on page 551

"NEWVAR" on page 565

OPENENTRYDIALOG

Displays a dialog window that lists catalog entries, and returns the user's selection

Category: Selection List

Syntax

selection-item=**OPENENTRYDIALOG**(<*type* <,*entry-name*<,*list-id*>>>);

selection-item
contains the four-level name of the selected SAS file, or a blank if nothing is selected.
Type: Character

type
specifies the member types to list in the dialog window, such as CLASS, SCL, or FRAME. This can reduce the length of the list in the dialog window. If *type* is not used, the names of all catalog entry types in the data library are listed.
Type: Character

entry-name
is a two- or four-level name of a catalog entry to call when a user selects the $\boxed{\text{Advanced}}$ button in the dialog window. The entry can be any of the following types: FRAME, SCL, PROGRAM, HELP, CBT, or MENU.
If *entry-name* is not specified, then the window does not contain $\boxed{\text{Advanced}}$.
Type: Character

list-id
contains the identifier of an SCL list that is passed to *entry-name*.
Type: Numeric or List

Details OPENENTRYDIALOG displays a list of SAS catalog entries from which a user can make a selection. You can pass that selection to code that opens the catalog entry.

If *entry-name* is supplied and $\boxed{\text{Advanced}}$ is selected, the item selected in the dialog window is passed to *entry-name* through the following variables, which are created automatically:

USERDATA
contains the list that is passed in the call to the OPENENTRYDIALOG and SAVEENTRYDIALOG functions.

LIBREF
contains the libref of the selected item.

MEMBER
contains the member name of the selected item.

ENTRY
contains the entry name of the selected item.

TYPE
contains the member type of the selected item.

List-id enables information to be passed between the calling entry and *entry-name*. It is passed to *entry-name* through the USERDATA variable.

Examples

Example 1: Opening a Catalog Entry Select a catalog entry of type FRAME to open and display the selected entry.

```
selection=openentrydialog('frame');
call display(selection);
```

Example 2: Using the Advanced Button with an SCL List Specify an entry to call when a user selects $\boxed{\text{Advanced}}$, as well as an SCL list that contains values to pass to the entry. The entry can be any of the following types: FRAME, SCL, PROGRAM, HELP, CBT, or MENU. If *entry-name* is not specified, then the window does not contain $\boxed{\text{Advanced}}$.

```
dcl list mydata;
mydata=makelist();
rc=insertc(mydata,'test');
selection=openentrydialog('frame',
   'mylib.mycat.myentry.frame',mydata);
if sysrc(1)=-1 then do;
   ...SCL statements to execute when
   the user cancels from the window...
   end;
else do;
   ...SCL statements to handle selections...
end;
```

The SCL entry for MYLIB.MYCAT.MYENTRY.FRAME contains the following program:

```
dcl char(8)  libref type;
dcl char(32) catalog entry;
dcl list userdata;
init:
put libref=;
put catalog=;
put entry=;
put type=;
call putlist(userdata);
return;
```

Note: The SCL entry must declare these exact variables, in the order specified, to properly reference the selected entry. △

If the user selects MYLIB2.MYCAT2.MYENTRY2.FRAME in the dialog window and then selects $\boxed{\text{Advanced}}$, the output for this program is

```
libref=Mylib2
catalog=Mycat2
entry=Myentry2
type=Frame
('test')[1]
```

See Also

"SAVEENTRYDIALOG" on page 627

OPENSASFILEDIALOG

Displays a dialog window that lists SAS files, and returns the user's selection

Category: Selection List

Syntax

selection=**OPENSASFILEDIALOG**(<*type*<,*level-count*<,*entry-name*<,*list-id*>>>>);

selection
contains the two- or three-level name of the selected SAS file, or a blank if nothing is selected.
Type: Character

type
specifies one or more member types to list in the dialog window, such as DATA, VIEW, MDDB, and CATALOG. This can reduce the length of the list in the dialog window. If *type* is not used, the names of all SAS files in the data library are listed. Multiple types should be separated by blanks.
Type: Character

level-count
specifies whether the function returns a two- or three-level name. The only valid choices are 2 and 3. The default is 2.
Type: Numeric

entry-name
is a two- or four-level name of a catalog entry to call when a user selects the Advanced button in the dialog window. The entry can be any of the following types: FRAME, SCL, PROGRAM, HELP, CBT, or MENU.
If *entry-name* is not specified, then the window does not contain Advanced.
Type: Character

list-id
contains the identifier of an SCL list that is passed to *entry-name*.
Type: Numeric or List

Details OPENSASFILEDIALOG displays a list of SAS files from which a user can make a selection. You can pass that selection to code that opens the SAS file.
If *entry-name* is supplied and Advanced is selected, the item selected in the dialog window is passed to *entry-name* through the following variables, which are created automatically:

USERDATA
contains the list that is passed in the call to OPENSASFILEDIALOG.

LIBREF
contains the libref of the selected item.

MEMBER
contains the member name of the selected item.

TYPE
contains the member type of the selected item.

List-id enables information to be passed between the calling entry and *entry-name*. It is passed to *entry-name* through the USERDATA variable.

Examples

Example 1: Opening a SAS File Select a SAS file of type DATA to open and display the table.

```
selection=opensasfiledialog('data');
call fsview(selection);
```

Example 2: Returning a Two-Level Name Open a SAS file of type DATA that returns a two-level name.

```
selection=opensasfiledialog('data view',2'work.a.a.scl',listid);
```

Example 3: Using the Advanced Button with an SCL List Specify an entry to be called when a user selects Advanced as well as an SCL list that contains values to pass to the entry. The entry can be any of the following types: FRAME, SCL, PROGRAM, HELP, CBT, or MENU. If *entry-name* is not specified, then the window does not contain Advanced.

```
dcl list mydata;
mydata=makelist();
rc=insertc(mydata,'test');
selection = opensasfiledialog('data',
   'mylib.mycat.myentry.frame',mydata);
if sysrc(1)=-1 then do;
   ...SCL statements to execute when
   the user cancels from the window...
end;
else do;
   ...SCL statements to handle selections...
end;
```

The SCL entry for MYLIB.MYCAT.MYENTRY.FRAME contains the following program:

```
dcl char(8) libref type;
dcl char(32) member;
dcl list userdata;
init:
put libref=;
put member=;
put type=;
call putlist(userdata);
return;
```

Note: The SCL entry must declare these exact variables, in the order specified, to properly reference the selected entry. △

If the user selects MYLIB2.MYMEMBER2.DATA in the dialog window and then selects Advanced, the output for this program is

```
libref=Mylib2
member=Mymember2
type=Data
('test')[1]
```

See Also

"SAVESASFILEDIALOG" on page 631

OPTGETC and OPTGETN

Return the current setting of a SAS system option

Category: SAS System Option

Syntax

cval=**OPTGETC**(*option-name*);

nval=**OPTGETN**(*option-name*);

cval
contains the setting of the SAS system option returned by OPTGETC, or a blank if *option-name* is invalid.
Type: Character

nval
contains the setting of the SAS system option returned by OPTGETN, or a missing value if *option-name* is invalid. For options with binary settings of ON or OFF, the function returns 1 if the setting is on, and 0 if the setting is off.
Type: Numeric

option-name
is the name of the SAS system option to retrieve.
Type: Character

Details If you try to use OPTGETC to get information about a numeric option or OPTGETN to get information about a character option, an error message is generated. To determine whether a SAS system option has a numeric or character setting, see the SAS documentation for your operating environment or *SAS Language Reference: Dictionary*.

For options that have ON/OFF settings (for example, DATE/NODATE), use OPTGETN, because these options have the numeric value 1 for ON and 0 for OFF.

You can view current option settings by using the OPTIONS procedure in the SAS session or by using the OPTIONS command.

Examples

Example 1: Using the OPTGETC Function Check to see whether new SAS tables will be compressed:

```
if optgetc('compress')='YES' then
    _msg_='Observations are compressed.';
else _msg_='Observations are not compressed.';
```

Example 2: Using the OPTGETN Function Return the setting of the CAPS option and place it in the column CAPS. The value returned is 0 if the NOCAPS form of the option is in effect, or 1 if CAPS is in effect.

```
caps=optgetn('caps');
```

See Also

"OPTSETC and OPTSETN" on page 584

OPTSETC and OPTSETN

Assign a value to a SAS system option

Category: SAS System Option

Syntax

rc=**OPTSETC**(*option-name*,*cval*);

rc=**OPTSETN**(*option-name*,*nval*);

rc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

option-name
is the name of the character SAS system option to set.
Type: Character

cval
is the new character setting for the option.
Type: Character

nval
is the new numeric setting for the option. For options with binary settings of ON and OFF, specify an *nval* of 1 to turn the option on and 0 to turn it off.
Type: Numeric

Details If you try to use OPTSETC to assign a value to a numeric option or OPTSETN to assign a value to a character option, an error message is generated. To determine whether a SAS system option has a numeric or character setting, see the SAS documentation for your operating environment or *SAS Language Reference: Dictionary*.

For options that have ON/OFF settings (for example, DATE/NODATE), use OPTSETN, because the options have the numeric values 1 for ON and 0 for OFF.

You can view current option settings by using the OPTIONS procedure in the SAS session or by using the OPTIONS command.

Examples

Example 1: Using the OPTSETC Function Set the COMPRESS option to allow compression of new SAS tables:

```
rc=optsetc('compress','yes');
```

Example 2: Using the OPTSETN Function Turn on the CAPS option:

```
rc=optsetn('caps',1);
```

See Also

"OPTGETC and OPTGETN" on page 583

PACKAGE

Defines a group of classes whose metadata must be recognized by objects defined on the client

Category: Object Oriented

Syntax

PACKAGE *package-name*<EXTENDS *parent-package-name*>;
    <*ITEM-statements*>
    ENDPACKAGE;

package-name
is the name that you want to assign to the class package. You can specify a name as a one- to four-level name.

parent-package-name
specifies the parent class package of *package-name*. *Parent-package-name* can be specified as a one- to four- level name.

ITEM-statements
are the ITEM statements that define the classes need by client objects. See "ITEM" on page 496 for more information.

Details

To build client/server applications, class information such as methods and attributes that are defined on the server must be recognized by objects that are defined on the client. The class package block defines a set of classes that are defined by the server. These classes can then be translated (by the WEB/AF proxy wizard or the CREATESCL function, for example) to interface formats and used by client objects to invoke methods and access attributes defined in these classes.

When you compile the entry that contains the PACKAGE block, SCL generates a CLASSPKG entry with the name that you specified in the PACKAGE statement.

No SCL statements, except for the IMPORT statement, are allowed within the SCL entry that contains the PACKAGE block.

Unlike RESOURCE entries, which directly embed class information in the entry, CLASSPKG entries use a reference (or link) to the class name of each of the classes that are defined in the class package block. Therefore, the class package will automatically pick up any changes that are made to the classes.

Example

If you enter the following code into an SCL entry, then issue the SAVECLASS command, SCL generates the entry **work.a.a.classpkg**.

```
package work.a.a.classpkg;
   item sashelp.classes.scllist.class;
   item sashelp.classes.format_c.class;
endpackage;
```

You can use the IMPORT statement to define the searchpath for CLASS references:

```
import sashelp.classes;
package work.a.a.classpkg;
   item scllist.class;
   item format_c.class;
endpackage;
```

See Also

"ITEM" on page 496

"ENDPACKAGE" on page 367

PATHNAME

Returns the physical name of a SAS data library or an external file

Category: External File

Syntax

filename=**PATHNAME**(*fileref*);

filename
contains the physical name of an external file or a SAS data library, or a blank if *fileref* is invalid.
Type: Character

fileref
is the fileref assigned to an external file or a SAS data library.
Type: Character

Examples

Example 1: Using PATHNAME with the FILEREF Function Use the FILEREF function to verify that the fileref MYFILE is associated with an external file, and then use PATHNAME to retrieve the name of the external file:

```
rc=fileref('myfile');
if (rc=0) then do;
     fname=pathname('myfile');
     put "Path = " fname;
end;
```

Example 2: Appending to a Pathname This example assigns the fileref F1 to **/u/sasssf**, and then uses F1 in a second call to FILENAME. The second call to FILENAME defines the fileref F2 by appending **/temp** to the path for F1.

```
rc=filename('f1','/u/sasssf');
path=pathname('f1');
put path=;
rc=filename('f2','temp','','','f1');
path=pathname('f2');
put path=;
```

The output would be

```
path=/u/sasccf
path=/u/sasccf/temp
```

Example 3: Concatenating Pathname In this example, FILENAME concatenates the directories **/u/sasccf** and **/u/sasshh** and assigns the concatenated list of names to the fileref F1. The second call to FILENAME is the same as the second call to FILENAME in Example 2. However, in this example, because F1 is assigned to a concatenated list of directory names instead of to a single directory, FILENAME does not append **/temp** to the paths.

```
rc=filename('f1','''/u/sasccf','/u/sasshh''');
path=pathname('f1');
put path=;
rc=filename('f2','temp','','','f1');
path=pathname('f2');
put path=;
```

The output would be

```
path=('/u/sasccf','/u/sasshh')
path=('/u/sasccf','/u/sasshh')
```

See Also

"FILENAME" on page 403

"FILEREF" on page 406

"FEXIST" on page 392

"FILEEXIST" on page 400

PICCLOSE

Closes the graphics environment

Category: Image

Syntax

rc=**PICCLOSE**(*graphenv-id*);

rc
contains the return code for the operation:

0	successful

>0	not successful

Type: Numeric

graphenv-id

contains the graphics environment identifier that was returned by PICOPEN.

Type: Numeric

Details PICCLOSE closes the graphic environment that was initialized by PICOPEN.

Example

Close the graphics environment that was opened by a previous PICOPEN:

```
fseterm:
   rc=picclose(graphenv_id);
return;
```

PICDELETE

Deletes a region

Category: Image

Syntax

rc=**PICDELETE**(*graphenv-id*,*region-id*);

rc

contains the return code for the operation:

0	successful
>0	not successful

Type: Numeric

graphenv-id

contains the graphics environment identifier that was returned by PICOPEN.

Type: Numeric

region-id

contains the region identifier that was returned by PICFILL

Type: Numeric

Example

Delete a region:

```
term:
   if (region_id ne 0 and graphenv_id ne 0) then
      rc=picdelete(graphenv_id,region_id);
return;
```

PICFILL

Defines a region and fills it with an image

Category: Image

Syntax

region-id=**PICFILL**(*graphenv-id*,*type*,*ulr*,*ulc*,*lrr*,*lrc*, *source*,<*command*<,*arguments*>>);

rc
contains the return code for the operation:

0	indicates that an error condition occurred.
>0	is the number identifying the region that was just defined.

Type: Numeric

graphenv-id
contains the graphics environment identifier that was returned by PICOPEN.
Type: Numeric

type
is the type of item to be displayed: IMAGE, GRSEG, or BITMAP.
Type: Character

ulr
is the upper left row of the screen area in which to display the image or graphics segment.
Type: Numeric

ulc
is the upper left column of the screen area in which to display the image or graphics segment.
Type: Numeric

lrr
is the lower right row of the screen area in which to display the image or graphics segment.
Type: Numeric

lrc
is the lower right column of the screen area in which to display the image or graphics segment.
Type: Numeric

source
specifies the location of the image.
For *type* BITMAP, specify the numeric value of a bitmap in a host-dependent resource file.
For *type* GRAPH, specify the four-level name of the graphic segment to be displayed.
For *type* IMAGE, specify either the name of an external file to be loaded or the path string from the LNAMEMK function.
Type: Numeric or Character

commands
lists commands that are used with *type* IMAGE to manipulate the image before displaying it. Valid commands are listed in “Commands Used with PICFILL” on page 590.
Type: Character

arguments
are arguments for the specified command.
Type: Numeric

Commands Used with PICFILL

For detailed information about these commands, see Appendix 1, “Commands Used with the IMGCTRL, IMGOP and PICFILL Functions,” on page 735 .

CROP
crops the selected image.

DITHER
dithers an image to a color map.

DITHER_BW
dithers the selected image to a monochrome black and white image.

EXECLIST
executes a list of commands.

GAMMA
applies a gamma value to the selected image.

MAP_COLORS
maps colors to the closest color in the selected color map.

MIRROR
mirrors an image.

NEGATE
changes the image to a negative.

QUANTIZE
reduces the number of colors used for an image.

ROTATE
rotates an image clockwise by 90, 180, or 270 degrees.

SCALE
scales an image.

Details PICFILL performs two functions:

- It defines the four corners of the region in which the image is to be displayed.
- It fills the region with the specified image.

If you specify a *command*, the PICFILL manipulates the image before displaying it.

Example

Display an image when the window opens:

```
init:
   imgfile=lnamemk(2,location,file);
   region_id=picfill(graphenv_id,"IMAGE",
                           15,12,40,78,imgfile);
return;
```

PICOPEN

Initializes the graphics environment

Category: Image

Syntax

graphenv-id=**PICOPEN**(*pane-number*);

graphenv-id
contains the identifier for the graphics environment:

0	indicates that an error condition occurred.
>0	is the graphics environment identifier for use in other PIC functions.

Type: Numeric

pane-number
is the number of the pane to contain the graphics environment:

1	the main portion of a SAS/AF window, or the first screen in an FSEDIT window.
2	the extended table portion of a SAS/AF window, or the second screen in an FSEDIT window.
n	the *n*th screen in an FSEDIT window.

Type: Numeric

Details The PICOPEN function initializes the graphics environment within a window. *Panes* are the physical areas of the screen in which text can be displayed. In SAS/AF software, there can be two panes per window, numbered 1 and 2. An FSEDIT window can have up to 101 screens.

A graphic environment must be opened for each pane in which an image or a graph is to be displayed.

Example

Initialize the graphics environment in the main portion of the FSEDIT window:

```
fseinit:
   graphenv_id=picopen(1);
return;
```

PMENU

Changes the PMENU for an application

Category: Window

Syntax

rc=**PMENU**(*pmenu*);

rc
contains the return code for the operation:

0	The PMENU entry was assigned successfully. However, the PMENU function does not verify that *pmenu* exists.
≠0	The operation was not successful.

Type: Numeric

pmenu
is the PMENU entry to assign. If the PMENU entry resides in a different catalog, then specify a value of the form *libref.catalog.pmenu*. If a one-level name is specified, then SCL searches for the PMENU entry first in the current catalog, then in SASUSER.PROFILE, and then in SASHELP.FSP.
Type: Character

Details The PMENU function only changes the pull-down menu that is associated with the window. It does not turn on the pull-down menus. If the pull-down menus are on, the new pull-down menu is displayed immediately.

See *SAS Procedures Guide* for more information about creating custom PMENU entries. FRAME entries can also use the _setPmenu method.

Example

Change the default PMENU that is associated with an application to MYPMENU in the catalog NEWLIB.TESTS:

```
rc=pmenu('newlib.tests.mypmenu');
```

POINT

Locates a row that is identified by the NOTE function

Category: SAS Table

Syntax

sysrc=**POINT**(*table-id*,*note-id*);

sysrc
contains the return code for the operation:

0	successful

≠0	not successful

Type: Numeric

table-id
is the identifier that was assigned when the table was opened. If *table-id* is invalid, the program halts.
Type: Numeric

note-id
is the identifier that was assigned to the row by the NOTE function, or −1 to go to the previous non-deleted row in the table. If *note-id* is invalid, the program halts and sends a message to the log.
Type: Numeric

Details POINT locates the row identified by *note-id*, which is a value that is returned from NOTE. The Table Data Vector is not updated until a "read" is performed by FETCH or FETCHOBS.

Example

Call NOTE to obtain a row ID for the last row that was read in the SAS table MYDATA. Call POINT to point to the row that corresponds to *note-id*. Call FETCH to return the row that is marked by the pointer.

```
dsid=open('mydata','i');
rc=fetch(dsid);
noteid=note(dsid);
   ...more SCL statements...
rc=point(dsid,noteid);
rc=fetch(dsid);
   ...more SCL statements...
rc=close(dsid);
```

See Also

"DROPNOTE" on page 361

"NOTE" on page 570

POPC, POPL, POPN, and POPO

Removes an item from an SCL list and returns the value of the item

Category: List

Syntax

cval=**POPC**(*list-id*<,*index*>);

sublist-id=**POPL**(*list-id*<,*index*>);

nval=**POPN**(*list-id*<,*index*>);

obj-id=**POPO**(*list-id*<,*index*>);

cval
contains the character value that was removed from the list by POPC.
Type: Character

sublist-id
contains the identifier of the sublist that was removed from the list by POPL.
Type: List

nval
contains the numeric value that was removed from the list by POPN.
Type: Numeric

obj-id
contains the identifier of the object item that was removed from the list by POPO.
Type: Object

list-id
contains the identifier of the list from which the value or sublist is removed. An invalid *list-id* produces an error condition.
Type: List

index
is the position of the item in the list. The position can be specified as a positive or negative number. By default, *index* is 1 (the first item). If *index* is a positive number, then the item is at position *index* from the beginning of the list. If *index* is a negative number, then the item is at position ABS(*index*) from the end of the list. An error condition results if the absolute value for *index* is zero or if it is greater than the number of items in the list.
Type: Numeric or List

Details These functions are useful for implementing stacks and queues of values.
An error condition results if

- the item has the NODELETE attribute
- the list has the FIXEDLENGTH or NOUPDATE attribute
- the list is empty
- the specified item is not a character item and you use POPC, the item is not a sublist and you use POPL, the item is not numeric and you use POPN, or the item is not an object and you use POPO.

To check the attributes of a list or list item, use HASATTR. To change attributes, use SETLATTR. Use ITEMTYPE to test the type of an item when the list contains items with types other than the one for which you are searching.

Examples

These examples assume that all list items are character. Programs A and B are equivalent. Both remove the last item in an SCL list.

```
A:  cval1=popc(listid,-1);
    put cval1=;
B:  cval2=getitemc(listid,-1);
    put cval2=;
    listid=delitem(listid,-1);
```

Example 1: Using the POPC Function This example creates an SCL list called TODOQ, which represents a queue of tasks to do. A SAS/AF FRAME entry has a text entry named NEWTASK for entering a new task into the TODOQ queue. A second text entry, TODO, displays the first task in the to-do queue. A DONE button removes the top task from the TODOQ queue.

```
INIT:
   todoq = makelist();
   done._gray();
return;

NEWTASK:
   todoq = insertc(todoq, newtask.text, -1);
      /* Enqueue */
   newtask.text = '';
   cursor newtask;
return;

DONE:
       /* Dequeue */
   finished = popc(todoq);
return;

MAIN:
   if listlen(todoq) then do;
      done._ungray();
      todo.text = getitemc(todoq);
   end;
   else do;
      done._gray();
      todo.text = '';
   end;
return;

TERM:
   rc = dellist(todoq);
return;
```

Example 2: Using the POPL Function This program searches for, retrieves, and deletes the first sublist item from the list LISTID:

```
LOOP:
   do i=1 to listlen(listid);
      if itemtype(listid,i)='L' then
         do;
            list=popl(listid,i);
            leave loop;
         end;
   end;
   ...other SCL statements...
```

Example 3: Using the POPN Function This example creates a new list called DATETIMES and treats it as a stack. The entry displays SAS datetime values when the button PUSH is pressed. The text entry pops and displays SAS datetime values from the DATETIMES stack when the button POP is pressed.

```
INIT:
   datetimes = makelist();
   pop._gray();
```

```
return;

PUSH:
   datetime.value = datetime();
   datetimes = insertn(datetimes, datetime.value);
   pop._ungray();
return;

POP:
   datetime.value = popn(datetimes);
   if listlen(datetimes) = 0 then
      pop._gray();
return;

TERM:
   rc = dellist(datetimes);
return;
```

Example 4: Using the POPO Function This example inserts an object into an SCL list and then removes the object from the list:

```
DCL sashelp.fsp.object.class obj1;
DCL object obj2;
init:
   obj1=_new_ sashelp.fsp.object.class();
   l1 = makelist();
   l1 = inserto(l1,obj1);
     /* Insert object as first item of list */
   obj2 = popo(l1);
     /* Remove object from the list into obj2 */
return;
```

See Also

"DELITEM" on page 335

"GETITEMC, GETITEML, GETITEMN, and GETITEMO" on page 442

"INSERTC, INSERTL, INSERTN, and INSERTO" on page 482

POPLEGEND

Restores to the LEGEND window the last contents saved with the PUSHLEGEND routine

Category: Legend

Syntax

CALL POPLEGEND();

Details POPLEGEND is useful if several entries in a CALL DISPLAY nested sequence have their own legends and you want to restore the original legend of each calling entry when a called entry ends.

If no legend contents have been saved, then this function is ignored.

To save the contents of the LEGEND window, use PUSHLEGEND.

For an example that includes POPLEGEND as well as other functions that manipulate a legend window, see "LEGEND" on page 507.

See Also

"ENDLEGEND" on page 366

"LEGEND" on page 507

"PUSHLEGEND" on page 603

"PUTLEGEND" on page 606

POPMENU

Displays a pop-up menu that contains character items from an SCL list

Category: List

Syntax

index=**POPMENU**(*list-id*<,*max-popup*<,*row*,*column*>>);

index
specifies the *index* of the item to be selected by default when the pop-up menu appears. On return, *index* will be set to the index of the selected pop-up menu item.
Type: Numeric

list-id
is the identifier of the list that contains the character or list items to display in the pop-up menu. An invalid *list-id* produces an error condition.
Type: Numeric or List
Lists may contain sublists as items. These list items must include one or more of the following named items to be valid:

Text
specifies the menu text (required).
Type: Character

Checked
specifies whether a check should appear to the left of the item. Valid values are Yes | No (Default).
Type: Character

Classifier
specifies the numeric representation of an icon to be associated with the menu item.
Type: Numeric

Grayed
specifies whether the menu item should be grayed. Valid values are Yes|No (Default).
Type: Character

HelpText
is the text to display on the status line.
Type: Character

Mnemonic
specifies the character that is used as a keyboard shortcut. The character must be one of the characters of the menu text.
Type: Character

max-popup
is the maximum number of items to display in a pop-up menu window. If the list is longer than *max-popup*, the menu is displayed in a window with a scrollable listbox.
Type: Numeric

row
is the starting row of the pop-up menu.
Type: Numeric

column
is the starting column of the pop-up menu.
Type: Numeric

Details If the list contains too many items to fit in a pop-up menu, POPMENU displays the choices in a list box that includes scrollbars. If no display window is available, POPMENU always puts the menu in a list box.

An item in the list that has the INACTIVE attribute cannot be selected and is grayed on devices that support graying. However, it is still displayed in the menu. You can use SETLATTR to assign the INACTIVE attribute to an item or to restore the item to an ACTIVE state.

If the pop-up menu is displayed in a list box, inactive items are marked with a dash, indicating that they may not be selected.

To display a separator on the pop-up menu, insert a dash as a character item in the list where you would like the separator to appear.

To display a dash, add a space before or after the dash to prevent the text from being interpreted as a separator.

POPMENU returns 0 if

- the user cancelled the pop-up menu or closed it without making a selection
- the list is empty
- the list contains numeric or sublist items
- there is insufficient memory to create the pop-up.

In the last three cases, the pop-up menu is not displayed.

If *row* and *column* are specified and either is outside the boundaries of the current window, the pop-up menu is positioned elsewhere on the window.

Note: Some window systems do not allow row and column positioning of pop-up menus. They may appear at the position where the pointing device was last clicked. △

Example

Display a pop-up menu when the user presses enter. Initialize the pop-up menu with three menu items: **`Select 1`**, **`Numeric`**, and **`Character`**. Define a separator between **`Select 1`** and **`Numeric`**.

```
init:
  dcl list ;
  control enter;
  list = { {text='Select 1',
               helpText='This is a selection.',
               mnemonic='S',
               classifier = 107},
          ''-'',
          ''Numeric'',
          ''Character''};

   return;
main:
 rc= popmenu (list);
 put rc=;
 return;

term:
/* Delete the list recursively to avoid memory leak */
rc = dellist ( list, 'y;);
return;
```

See Also

"FILLIST" on page 407

"SETLATTR" on page 656

PREVIEW

Manipulates an application's preview buffer

Category: Submit Block

Syntax

rc=**PREVIEW**(*action*<,*argument-1*<,*argument-2*<,*argument-3*>>>);

rc
contains the return code for the operation:

0	The operation was successful.
–1	For actions that open the PREVIEW window, the CANCEL command was used to exit the PREVIEW window. For other actions, the requested operation was not performed successfully.

Type: Numeric

action
is an action from "Actions for the PREVIEW Function" on page 600.
Type: Character

argument
specifies up to three arguments for the specified action from "Actions for the PREVIEW Function" on page 600.
Type: Character

Actions for the PREVIEW Function

'BROWSE'
opens the PREVIEW window for browsing only. You can optionally specify a window title as *argument-1*.

'CLEAR'
clears the preview buffer. Any statements that were generated previously by SUBMIT statements or that were included by COPY or INCLUDE actions are lost. No optional arguments are used with this action.

'CLOSE'
closes the PREVIEW window. No optional arguments are used with this action.

'COPY'
copies a catalog entry of type SOURCE, OUTPUT, LOG, or SCL into the preview buffer. You must specify the name of the entry to be copied as *argument-1*. For the syntax for *argument-1*, see the description of the SAVE action.

Note: If you copy an SCL entry to the PREVIEWwindow, you should CLEAR the window before closing it. Otherwise, the SCL code that was copied will be submitted to SAS. △

'DISPLAY | EDIT'
opens the PREVIEW window to display the contents of the preview buffer for editing. All standard text editor commands are valid in this window. Control stays with the PREVIEW window until the PREVIEW window is exited. Issue the END command to exit the window and return to the SAS/AF program. Changes that a user makes to the statements in the window are not reversed by issuing a CANCEL command. However, the SCL program can check for the return code of –1 from the PREVIEW function, indicating that a CANCEL command was issued in the PREVIEW window, and can then specify not to save the contents of the PREVIEW window.
You can optionally specify a window title as *argument-1*.

'FILE'
saves the current contents of the PREVIEW window to an external file. With this action, you must specify the fileref for the external file as *argument-1*. You must specify a fileref; physical filenames are not allowed.
You can specify APPEND for *argument-2* to append the contents of the preview buffer to the contents of the external file.

'HISTORY'
saves or appends the statements submitted from SCL programs to a catalog member. You must specify the name of an entry as *argument-1*, in one of the following forms:

entry
saves the submitted statements in SASUSER.PROFILE.*entry*.SOURCE.

entry.SOURCE
saves the submitted statements in SASUSER.PROFILE.*entry*.SOURCE.

libref.catalog.entry
saves the submitted statements in *libref.catalog.entry*.SOURCE.

libref.catalog.entry.SOURCE
saves the submitted statements in *libref.catalog.entry*.SOURCE.

Once a history destination is set, it remains in effect for the application until a reset is performed. To clear the destination, invoke PREVIEW again by passing the HISTORY action without any subsequent argument. The application stops appending the submitted statements to the previously specified SOURCE entry.

You can also optionally specify a description of up to 40 characters for the entry as *argument-2*.

'INCLUDE'
copies into the PREVIEW window the contents of an external file. With this action, you must specify the fileref for the external file as *argument-1*. A physical filename is not allowed.

'LINES'
returns the number of the last non-blank line in the PREVIEW window.

'MODIFIED'
returns 1 if the PREVIEW window was modified and 0 if it was not modified.

'OPEN'
displays the PREVIEW window and immediately returns control to the SCL program. New SAS statements are displayed continually as they are generated.

This window is displayed until the application closes or until you use the PREVIEW function and specify the CLOSE action. Thus, specifying OPEN enables you to keep the window open throughout an application by allowing control to return to the procedure while the window is open. With OPEN, you can optionally specify a window title as *argument-1*.

'PRINT'
prints the contents of the PREVIEW window. This action accepts three arguments:

- You can optionally use *argument-1* to specify a form to use for controlling printing. Use *name* or *name*.FORM to specify a form in the current catalog. Use *libref.catalog.name* or *libref.catalog.name*.FORM to specify a form in a different catalog. If you do not specify a form, SASHELP.FSP.DEFAULT.FORM is used. For more information about forms, see *SAS Language Reference: Dictionary*.
- You can optionally use *argument-2* to specify a fileref for a print file. If this argument is blank or missing, the printout is sent to the default system printer.
- You can optionally use *argument-3* to specify append mode. Use **A** to append the output to the print file. If this argument is blank or missing, each new PRINT option replaces the contents of the print file.

'SAVE'
copies the contents of the PREVIEW window to a catalog member. With this action, you must specify the name of an entry as *argument-1*, in one of the following forms:

entry
saves the contents in SASUSER.PROFILE.*entry*.SOURCE.

entry.type
saves the contents in SASUSER.PROFILE.*entry.type*. Entry types can be SOURCE, LOG, OUTPUT, or SCL.

libref.catalog.entry
saves the contents in *libref.catalog.entry*.SOURCE.

libref.catalog.entry.type
saves the contents in *libref.catalog.entry.type*.

You can also optionally specify a description for the entry as *argument-2*. The first 256 characters of the description are saved.

You can also optionally specify **APPEND** as *argument-3* to append the contents of the PREVIEW window to the specified catalog entry.

Details The preview buffer is where statements that are generated by SUBMIT blocks are stored before they are submitted for execution.

Example

Manipulate the PREVIEW window. The user enters values in text entry controls TABLENAME (the name of a SAS table to be created), MIMIC (the name of an existing SAS table after which the new table is modeled), and FNAME (the fileref of a file in which submitted SAS statements will be stored). If the user does not issue a CANCEL command from the application window, then PREVIEW displays the statements in the PREVIEW window. If the user does not issue a CANCEL command from the PREVIEW window, then the statements are submitted for execution. If a user issues a CANCEL command from the application or from the PREVIEW window, the PREVIEW window is cleared of all statements. When a user presses the RUN button, the statements are submitted.

```
INIT:
   control label;
return;
RUN:
 submit;
    data &tablename;
       set &mimic;
    run;
  endsubmit;
 if (_status_ ne 'C') then
    do;
     if (preview('EDIT') = -1) then
        rc=preview('clear');
       else
        do;
          rc=preview('FILE',FNAME);
            submit continue;
             endsubmit;
           end;
      end;
   else
      rc=preview('clear');
return;
```

See Also

"ENDCLASS" on page 366

"SUBMIT" on page 676

"WREGION" on page 730

PROTECT

Assigns protection to a FRAME entry control or field

Category: Control or Field

Syntax

PROTECT *wcol-names* | _ALL_;

wcol-names
lists one or more window columns to protect, or **_ALL_** to protect all window columns. To specify multiple window columns, separate the names with blanks.
Type: Character

Details Protecting a window column prevents the cursor from tabbing to the associated control or field. You can use the PROTECT statement to temporarily override a PROTECT attribute that has been specified for a window column. The column to be protected cannot be an element of an array. To protect an array element, use the FIELD function.

If you protect a field column with the PROTECT statement in FSEDIT applications and issue the MODIFY command to edit the custom screen, the PROTECT attribute is saved for that field column in the PROTECT window.

Window controls can also use the _protect method. Protecting some window controls (block, check box, icon, list box, pushbutton, radio box, scroll bar, and slider) is the same as calling the GRAY function or the _gray method.

The protection status remains in effect until the UNPROTECT statement is used.

The following statements are incorrect because they do not name window columns:

```
protect a{i};
protect a||b;
```

Example

Prevent the user from changing the value of the window column TABLENAME after the value has been entered:

```
if (modified(tablename) and ERROR(tablename)=0)
then protect tablename;
```

See Also

"DISPLAYED" on page 353

"ERROR" on page 375

"ERROROFF" on page 376

"ERRORON" on page 377

"FIELD" on page 394

"MODIFIED" on page 550

"UNPROTECT" on page 695

PUSHLEGEND

Saves the contents of the LEGEND window

Category: Legend

Syntax

CALL PUSHLEGEND();

Details PUSHLEGEND is useful if several entries in a CALL DISPLAY nested sequence have their own legends and you want to restore the original legend of each calling entry when a called entry ends.

To restore a pushed legend to the LEGEND window, use POPLEGEND.

For an example that includes PUSHLEGEND as well as other functions that manipulate a legend window, see “LEGEND” on page 507.

See Also

“ENDLEGEND” on page 366

“LEGEND” on page 507

“POPLEGEND” on page 596

“PUTLEGEND” on page 606

PUT

Writes text to the LOG window

Category: Utility

Comparisons: SAS Statement with limitations in SCL

Syntax

PUT <<'*character-string*'> <*variable-name*<=>><*object.attribute*<=>>> | _ALL_;

character-string
is the literal text to write to the LOG window.
Type: Character

variable-name
lists one or more variables whose name and value are to be written to the LOG window.
Type: Character

ALL
specifies that all variables and their values are to be written to the LOG window.
Type: Character

Details

PUT supports dot notation for returning the value of an object. Use the form **put object.attribute=;**. For more information about dot notation, see “Accessing Object Attributes and Methods With Dot Notation” on page 119.

Differences in PUT Statement Execution

SCL supports only the forms of the PUT statement shown in "Syntax." You can combine these forms. For example:

```
PUT 'character-string' variable-name=;
```

For details about the PUT statement in the base SAS language, see *SAS Language Reference: Dictionary* .

PUTC and PUTN

Return a formatted value, using the specified format

Category: Formatting

Syntax

formatted-val=**PUTC**(*char-val*,*format*);

formatted-val=**PUTN**(*num-val*,*format*);

formatted-val
contains the value with the specified format applied.
Type: Character

char-val
is the character value to be formatted by PUTC.
Type: Character

num-val
is the numeric value to be formatted by PUTN.
Type: Numeric

format
is the character format to apply with PUTC or the numeric format to apply with PUTN. *Format* can be an SCL variable or a character literal.
Type: Character

Details Dot notation cannot be used with the PUTC or PUTN functions. This restriction is necessary to allow proper parsing of the *char-val* and *num-val* parameters.

For more information about using a format to return a value, see the PUT function in *SAS Language Reference: Dictionary* .

Examples

Example 1: Using the PUTC Function Format the value that a user enters into the text entry control named VALUE:

```
MAIN:
   value=putc(value,'$QUOTE.');
   put value=;
return;
```

Entering SAS into the field displays "SAS" in the field and produces the following output:

```
VALUE="SAS"
```

Example 2: Using the PUTN Function Format the variable NETPD using the DOLLAR12.2 format, and store the value in the variable SALARY.

```
INIT:
   netpd=20000;
   put netpd=;
   fmt='dollar12.2';
   salary=putn(netpd,fmt);
   put salary=;
return;
```

This program produces the following output:

```
NETPD=20000
SALARY=  $20,000.00
```

See Also

"INPUTC and INPUTN" on page 481

PUTLEGEND

Specifies the contents of one line in the LEGEND window

Category: Legend

Syntax

CALL PUTLEGEND(*line*,*text*<,*color*<,*attribute*>>);

line
is the number for the line on which to display the text. If this value is larger than the MAXROW returned from the WINFO function, the line number is ignored.
Type: Numeric

text
is the text to display on one line of the LEGEND window. Once you specify text for a legend line, that text is redisplayed each time the LEGEND routine is called. To delete the text for a line, you can specify either new text or a null string (`''`)for that line number.
Type: Character

color
is a color name, or `''`for the default color. Colors are BLACK, BLUE, BROWN, CYAN, GRAY, GREEN, MAGENTA, ORANGE, PINK, RED, WHITE, and YELLOW. SASCOLOR window elements can also be used for *color*.
The default color is the SASCOLOR window element "Informational Text."
Type: Character

attribute
is a display attribute, or `''`for the default attribute. Attributes are: NONE, BLINKING, HIGHLIGHT, HIREV, REVERSE, and UNDERLINE. If you specify a SASCOLOR window element for *color*, then *attribute* is ignored, because the SASCOLOR window element contains a display attribute.
The default attribute is the SASCOLOR window element "Informational Text."
Type: Character

Details Use multiple PUTLEGEND routines to display multiple lines. The default legend window size allows four lines of text. You can change the LEGEND window size by using WREGION. To display the legend window, use LEGEND.

For an example that includes PUTLEGEND as well as other functions that manipulate a legend window, see “LEGEND” on page 507.

See Also

“ENDLEGEND” on page 366

“LEGEND” on page 507

“POPLEGEND” on page 596

“PUSHLEGEND” on page 603

PUTLIST

Displays the contents of an SCL list in the LOG window

Category: List

Syntax

CALL PUTLIST(*list-id*<,*label*<,*indent*>>);

list-id
is the identifier returned by the function that created the list.
Type: Numeric or List

label
specifies the label for the printed output.
Type: Character

indent
is the number of characters to indent list items in the printed list.
Type: Numeric

Details After printing the optional label, PUTLIST prints a left parenthesis '(' to mark the beginning of the list, followed by the list of items separated by blanks. Each named item is preceded by its name and an equal sign (=), but nothing is printed before items that do not have names. PUTLIST ends the list with a right parenthesis ')', followed by the list's identifier number within square brackets.

If the value for *indent* is greater than or equal to 0, the list is printed in a vertical format where each list item is printed on its own line. Sublists are indented the number of spaces to the right that is specified by *indent*.

If the list contains sublists that have been deleted, PUTLIST identifies each invalid list identifier with the text **`<invalid list id>[listid]`**.

Examples

The following examples are based on an SCL list whose list identifier is stored in the variable A. This list contains the numbers 17 and 328 plus the character value "Any characters". These examples display the list in several ways:

- in its simplest form
- with values indented
- after a numeric item is replaced with a sublist item
- after the list has been added to itself as a sublist.

Example 1: Displaying a List with Indented Values Print a list and indent the list items:

```
call putlist(a,'A=',2);
```

The above statement produces the following output:

```
A=(  17
     328
     'Any characters'
   )[7]
```

Example 2: Replacing a Numeric Item with a Sublist Item Replace the second item in the list A with the list identifier for sublist B, which contains the values -4.75 and 12.875:

```
/*  Assign the second item to list B.  */
   a=setiteml(a,b,2);
   name=nameitem(a,1,'MIN');
   name=nameitem(a,2,'B');
   call putlist(a,'A=',2);
```

These statements produce the following output:

```
A=(  MIN=17
  B=(  -4.75
       12.875
  )[5]
  'Any characters'
  )[7]
```

Example 3: Adding a List to Itself as a Sublist If a sublist appears more than once in the list that is being printed, PUTLIST prints only the following for the second and subsequent occurrences of the list:

```
(...) [listid-number]
```

To view the full contents of the list, scan the output of PUTLIST for other occurrences of [*listid-number*]. This prevents infinite loops if a list contains itself.

Create and display a recursive list:

```
r1=makelist();
   r1=setnitemn(r1,1,'X');
   r1=setniteml(r1,r1,'SELF');
   call putlist(r1,'R1=',2);
```

These statements display the following information in the LOG window. Note that the full contents of the list that has the identifier 7 are printed only once. The other occurrence is represented as (...)[7].

```
R1=(  X=1
      SELF=(...)[7]
   )[7]
```

See Also

"MAKELIST" on page 535

"MAKENLIST" on page 536

PUTVARC and PUTVARN

Write a value to the Table Data Vector (TDV) for a SAS table

Category: SAS Table

Syntax

CALL PUTVARC(*table-id*,*col-num*,*cval*);

CALL PUTVARN(*table-id*,*col-num*,*nval*);

table-id
is the identifier that was assigned when the table was opened. If *table-id* is invalid, the program halts.
Type: Numeric

col-num
is the number of the column in the SAS table. This is the number that is adjacent to the column when the CONTENTS procedure lists the columns in the SAS table. You can use the VARNUM function to obtain this value.
Type: Numeric

cval
is the character value to be written to the TDV.
Type: Character

nval
is the numeric value to be written to the TDV.
Type: Numeric

Details After PUTVARC writes a character value to a table column, use UPDATE to update the row in the SAS table.

If the SCL program uses CALL SET to link columns in the SCL data vector (SDV) with columns in the Table Data Vector (TDV), do not use the PUTVARN and PUTVARC routines for any columns that are linked by SET. UPDATE and APPEND automatically copy the data from the SDV to the TDV before writing a row to a physical file. Therefore, the value that is copied from the SDV will overwrite the value written to the TDV by PUTVARC or PUTVARN, and the value of the corresponding table column will not be updated with the value specified by PUTVARC or PUTVARN.

Examples

Example 1: Using the PUTVARC Routine Change an employee's last name from **SMITH** to **UPDIKE** in the column NAME in the table referenced by the table identifier PAYID:

```
vnum=varnum(payid,'name');
rc=locatec(payid,vnum,'SMITH','u');
if (rc>0) then
   do;
      call putvarc(payid,vnum,'UPDIKE');
      rc=update(payid);
   end;
```

Example 2: Using the PUTVARN Routine Change an item's price from 1.99 to 2.99 in the table referenced by the table identifier PAYID:

```
vnum=varnum(payid,'price');
rc=locaten(payid,vnum,1.99,'u');
if (rc>0) then
   do;
      call putvarn(payid,vnum,2.99);
      rc=update(payid);
   end;
```

See Also

"APPEND" on page 251

"FETCH" on page 389

"FETCHOBS" on page 390

"GETVARC and GETVARN" on page 450

"INITROW" on page 480

"UPDATE" on page 697

"VARNUM" on page 713

REDIM

Resizes a dynamic array

Category: Array

Syntax

rc=**REDIM**(*array,dim1<,dim2<,dim3....<,dimN>...>>*);

rc

indicates whether the operation was successful.

0	successful
≠0	not successful

Type: Numeric

array
is the dynamic array to be resized. A non-dynamic array causes an error condition.
Type: Array

dim1...dimN
is the size of each specified dimension. If you specify negative sizes or an invalid number of dimensions, an error condition occurs.
Type: Numeric

Details You can use the REDIM function to resize a dynamic array. You cannot change the numbers of dimensions or type of the array, only the bounds. The REDIM function will preserve the data in the array. However, if you resize the array to a smaller size, you will lose the data in the eliminated elements. There is no limit to the number of times that you can resize an array.

Examples

Example 1: Create an Array and Resize It Preserving the Data The example creates a one-dimensional array of 5 elements and resizes it, preserving the data, to 10 elements.

```
DCL num a(*);
rc = makearray(a,3);
do i=1 to dim(a);
   a[i]=i;
end;
rc = redim(a,5);
put a=;
rc=delarray(a);
```

The output would be:

```
a=
a[1]=1
a[2]=2
a[3]=3
a[4]=.
a[5]=.
```

Example 2: Create an Array and Resize it Without Preserving the Data The example creates a two-dimensional 5x5 array, resizes it to a 10x10 array and does not preserve the data.

```
DCL num a(*);
rc = makearray(a,3);
do i=1 to dim(a);
   a[i]=i;
end;
rc = makearray(a,5);
put a=;
rc = delarray(a);
```

The output would be:

```
a=
a[1]=.
a[2]=.
a[3]=.
a[4]=.
```

```
a[5]=.
```

See Also

"DELARRAY" on page 332

"MAKEARRAY" on page 533

Chapter 4, "SCL Arrays," on page 37

REFRESH

Redisplays a window using current field or control values

Category: Window

Syntax

REFRESH;

Details Refreshing a window can result in updating window variable values. FRAME entries can also use the _refresh method.

Example

Suppose the field NAME is displayed on the left side of a PROGRAM entry and that NEW.PROGRAM is sized so that it is displayed on the right side of the window. When the following code runs, NAME changes to red and then NEW.PROGRAM is displayed. If the REFRESH statement were not present, NAME would not appear red until NEW.PROGRAM is closed and control is returned to the application.

```
rc=field('color red','name');
refresh;
call display('new.program');
```

RENAME

Renames a member of a SAS data library, an external file, or a directory

Category: Utility

Syntax

sysrc=**RENAME**(*old-name,new-name*<,*type*<,*description*<,*generation*<,*password*>>>>);

sysrc
contains the return code for the operation:
Type: Numeric

0	successful
≠0	not successful

old-name
is the current name of a member of a SAS data library, an external file, or an external directory. For a member, this can be a one-, two-, or four-level name. For an external file or a directory, *old-name* must be the full pathname of the file or a directory; otherwise, the current directory is used.
Type: Character

new-name
is the new one-level name for the library member, external file, or directory.
Type: Character

type
specifies the type of element to rename:
Type: Character

'ACCESS'
an access descriptor that was created using SAS/ACCESS software

'CATALOG'
a SAS catalog or catalog entry

'DATA'
a SAS table (This is the default.)

'VIEW'
a SAS table view

'FILE'
an external file or a directory.

description
is the description of a catalog entry. You can specify *description* only when *type* is CATALOG.
Type: Character

password
is the password for the file that is being renamed.
Type: Character

generation
is the generation number of the data set that is being renamed.
Type: Numeric

Details You can use RENAME to rename files or directories that are external to a SAS session as well as members of a SAS data library.

To rename an entry in a catalog, specify the four-level name for *old-name* and a one-level name for *new-name*. You must specify CATALOG for type when renaming an entry in a catalog.

Examples

Example 1: Renaming Tables and Catalog Entries Rename a SAS table from DATA1 to DATA2. Also rename a catalog entry from A.SCL to B.SCL.

```
rc1=rename('mylib.data1','data2');
rc2=rename('mylib.mycat.a.scl','b','catalog');
```

Example 2: Renaming an External File Rename an external file:

```
   /* rename a file that is in another directory */
rc=rename('/local/u/testdir/first',
          '/local/u/second','file');
   /* rename a PC file */
rc=rename('d:\temp','d:\testfile','file');
```

Example 3: Renaming a Directory Rename a directory:

```
rc=rename('/local/u/testdir/','/local/u/oldtestdir','file');
```

Example 4: Renaming a Data Set Generation Rename a generation of the data set **work.one** to **work.two**, where the password for **work.one#003** is "blahblah":

```
rc=rename('work.one','two','data','','blahblah',3);
```

See Also

"DELETE" on page 334

REPLACE

Substitutes a replacement string for a reference to an SCL variable in the SUBMIT block

Category: Submit Block

Syntax

REPLACE *variable replacement-string*;

variable
is the variable whose value the replacement value is substituted for.
Type: Character

replacement-string
is the text to substitute for the variable's value. This text can include a variable's value, but that is not mandatory.
Type: Character

Details REPLACE substitutes a replacement string for a reference to an SCL variable in the SUBMIT block only if the variable is not blank. It functions as an implicit IF statement, determining when to substitute the string in the SUBMIT block. Using the REPLACE statement reduces the amount of code needed to generate statements to be submitted.

The REPLACE statement is evaluated when the program is compiled. Different replacement strings cannot be substituted based on conditions that exist at execution time. For example, the following statements cause errors when you compile the program:

```
if (x) then
      replace y '&y';
   else
      replace y '&z';
```

If you use multiple REPLACE statements for the same variable, the last REPLACE statement is used and a warning is generated by the compiler to that effect.

A good programming practice is to collect all the REPLACE statements in one place in your SCL program.

You can also use the REPLACE option in the ATTR window of a PROGRAM entry to specify the replacement string. However, this can be overridden by REPLACE statements in the SCL program.

SCL performs substitution according to the following rules:

- If the value of the SCL variable is blank (or _BLANK_), no substitution is performed.
- If the value of the SCL variable is not blank, SCL performs substitution into the replacement string for the variable and substitutes the resulting string into the SUBMIT block.

The replacement string can reference other SCL variables.

Note: Replacement strings are not recursive. When you refer to another variable in the replacement string, the program uses the current value of the variable, not the value that is based on its replacement string. △

Example

```
replace tablename 'data=&tablename';
   ...more SCL statements...
submit continue;
   proc print &tablename;
   run;
endsubmit;
```

If TABLENAME contains `''`(or _BLANK_), the submitted statements are

```
submit continue;
   proc print;
   run;
endsubmit;
```

However, if TABLENAME contains **`work.sample`**, the submitted statements are

```
submit continue;
   proc print data=work.sample;
   run;
endsubmit;
```

RETURN

Stops executing statements in the program section that is currently executing and may return a value to the caller

Category: Control Flow

Comparisons: SAS statement with limitations in SCL

Syntax

RETURN<(*value*)>;

value
is a value to be returned by the current method. *Value* can be an SCL variable, a numeric or character constant, or an expression (except for an array). Use *value* only for RETURN statements that end METHOD and ENTRY statements.
Type: Character, Numeric, List, Object

Details When RETURN stops executing statements in the current section of an SCL program, control passes to the next section in the program execution cycle.

The RETURN statement for an ENTRY or METHOD statement block can return *value* if the ENTRY or METHOD statement contains RETURN=*data type*. The returned value from the RETURN statement has no effect if the statement does not immediately return the program control back to the calling program.

For details about the RETURN statement in the base SAS language, see *SAS Language Reference: Dictionary*.

Example

Define a method for LIB.CAT.MYCLASS.CLASS that compares two lists, L1 and L2. Return 1 if the strings are identical and 0 if the strings are not identical.

```
useclass lib.cat.myclass.class;
compare: method l1 l2:list return=num;
  dcl num len1 len2 i;
  len1=listlen(l1);
  len2=listlen(l2);
  if (len1 ^= len2) then return 0;

  do i=1 to len1;
    dcl char(1) type;
    type=itemtype(l1,i);
    if (type ^= itemtype(l2,i)) then
      return 0;

    select type;
      when ('O') do;
        if (getitemo(l1,i) ^= getitemo(l2,i)) then
          return 0;
      end;
      when ('N') do;
        if (getitemn(l1,i) ^= getitemn(l2,i)) then
          return 0;
      end;
      when ('L') do;
        if (getiteml(l1,i) ^= getiteml(l2,i)) then
          return 0;
      end;
      when ('C') do;
        if (getitemc(l1,i) ^= getitemc(l2,i)) then
```

```
            return 0;
        end;
        otherwise return 0;
      end;
    end;
    return(1);
  endmethod;
  enduseclass;
```

See Also

"ENTRY" on page 369

"METHOD" on page 540

"RUN" on page 625

"STOP" on page 674

REVLIST

Reverses the order of the items in an SCL list

Category: List

Syntax

list-id=**REVLIST**(*list-id*);

list-id
is the identifier of the list to reverse. The function returns the list identifier that is passed in. An invalid *list-id* produces an error condition.
Type: Numeric or List

Details Any names and attributes that are assigned to list items remain with the items when the items are reversed.

REVLIST does not make a copy of the list before reversing the order of the list items. The list is modified in place. To keep a copy of the original list, use COPYLIST before REVLIST.

An error condition results if the list has the NOUPDATE attribute.

To check attributes, use HASATTR. To change attributes, use SETLATTR.

Example

Make a nonrecursive copy of the list identified by MYLISTID, reverse the items in the copied list, and assign the new list identifier to the variable REVLISTID:

```
revlistid = revlist(copylist(mylistid));
```

See Also

"HASATTR" on page 458

"ROTLIST" on page 622

"SETLATTR" on page 656

"SORTLIST" on page 669

REWIND

Positions the table pointer at the beginning of a SAS table

Category: SAS Table

Syntax

sysrc=**REWIND**(*table-id*);

sysrc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

table-id
is the identifier that was assigned when the table was opened in any mode except IS, US, or N. If *table-id* is invalid, the program halts.
Type: Numeric

Details After a call to REWIND, a call to FETCH reads the first row in the table.
If there is an active WHERE clause, REWIND moves the table pointer to the first row that satisfies the WHERE condition.

Example

Call FETCHOBS to fetch the tenth row in the table MYDATA. Then call REWIND to return to the first row and fetch the first row:

```
dsid=open('mydata','i');
rc=fetchobs(dsid,10);
rc=rewind(dsid);
rc=fetch(dsid);
```

See Also

"FETCH" on page 389

"FETCHOBS" on page 390

"FREWIND" on page 431

"NOTE" on page 570

"POINT" on page 592

RGBDM

Returns the name supported by the SAS windowing environment for a color

Category: Utility

Syntax

DM-color-name=**RGBDM**(*color-name*<,*RGB-color*>);

DM-color-name
contains the name of the SAS windowing environment color that is closest to *color-name*.
Type: Character

color-name
is a color name to look up: BLACK, BLUE, BROWN, CYAN, GRAY, GREEN, MAGENTA, ORANGE, PINK, RED, WHITE, or YELLOW. Any SAS/GRAPH color name is allowed, as well as SASCOLOR window elements. Arbitrary RGB colors can be specified using the CX*rrggbb* convention.
Type: Character

RGB-color
returns *color-name* in the CX*rrggbb* format.

Note: This parameter is an update parameter. See "Input, Output, and Update Parameters" on page 34 for more information. △
Type: Character

Details The RGBDM function provides a way to determine both the closest color that is supported by the SAS windowing environment for a specified color and the RGB values for a color name. If *color-name* is a variable and the specified color is a valid color abbreviation, the variable is updated with the complete color name. For example, the color "R" would be translated to "RED" and the *RGB-color* would be "CXFF0000". The value for *RGB-color* may be different depending on the host operating system.

For more information about the CX*rrggbb* format, see *SAS/GRAPH Software: Reference*.

Example

Display the color components of several different colors:

```
length rgbclr $ 8;
INIT:
```

```
         /* display txtclr=red */
         /* rgbclr=cxff0000    */
      txtclr = rgbdm("RED", rgbclr);
      put txtclr= rgbclr=;

         /* display txtclr=red */
      txtclr = rgbdm("cxf00000");
      put txtclr=;

         /* display txtclr=red clr=red */
      clr='r';
      txtclr = rgbdm(clr);
      put txtclr= clr= ;

         /* display the foreground */
         /* color in rgb values    */
      txtclr = rgbdm("FOREGROUND", rgbclr);
      put txtclr= rgbclr=;
   return;
```

In some operating environments, this example produces the following output:

```
txtclr=RED rgbclr=CXFF0000
txtclr=RED
txtclr=RED clr=RED
txtclr=WHITE rgbclr=CXF2F2F2
```

RIGHT

Returns a right-aligned character value

Category: Character

Syntax

rstring=**RIGHT**(*string*<,*length*>);

rstring
contains the right-aligned character string. If *rstring* already exists, then specifying a length in the RIGHT function changes the current length of *rstring* to *length*. If the length has not been defined with a DECLARE or LENGTH statement, and if *rstring* is not a window variable, then the default SCL variable length is 200 characters.
Type: Character

string
is the character string to be right-justified.
Type: Character

length
is the length in which the character string is to be right-justified. The default is the maximum length of *rstring*.
Type: Numeric

Details The string is justified by padding with leading spaces. The default length is the maximum length of *rstring* characters.

In order for RIGHT to work properly when *rstring* is a window variable, set the justification attribute (**JUST**) for *rstring* to **NONE** when you define the window variables.

To left-justify a character string, use the LEFT function when you define the window variable. To center a character string, use CENTER.

See Also

"CENTER" on page 275

"LEFT" on page 506

RLINK

Reports whether a link exists between the current SAS session and a remote SAS session

Category: Interface to SAS Software

Requires SAS/CONNECT software

Syntax

rc=**RLINK**(*remote-session-id*);

rc

contains the return code for the operation:

1	The link exists.
0	The link does not exist.

Type: Numeric

remote-session-id

is the name of the remote session (REMOTE= value) that is being tested.

Type: Character

Details See *SAS/CONNECT User's Guide* for details about accessing remote hosts from SAS software.

To get the name of the last remote host that was linked to during the current SAS session, use OPTGETC, specifying 'REMOTE' as *option-name*.

Example

Check to see whether the link is active:

```
REMSESS=optgetc('remote');
msg=sysmsg();
put msg REMSESS;
rc=rlink(REMSESS);
   if (rc=0) then
      msg='No link exists.';
   else
      msg='A link exists.';
```

```
put msg;
```

See Also

"OPTGETC and OPTGETN" on page 583

"RSESSION" on page 623

"RSTITLE" on page 624

ROTLIST

Rotates the items in an SCL list

Category: List

Syntax

list-id=**ROTLIST**(*list-id*<,*n*>);

list-id
is the identifier of the list to rotate. The function returns the list identifier that is passed in. An invalid *list-id* produces an error condition.
Type: Numeric or List

n
is the number of times to rotate the list. The default is 1.
Type: Numeric

Details The items are rotated the number of times specified by *n*. If the value for *n* is positive, the items are rotated from right to left. This means that each rotation moves the item at the front of the list to the end of the list (that is, from position 1 to position -1). If the value for *n* is negative, the items are rotated from left to right. This moves the item at the end of the list to the front of the list (that is, from position -1 to position 1).

When a list is rotated, item names and attributes are moved along with the elements.

Fetching a named item from a list that has more than one item of the same name may return a different item from the list after rotating than was returned before rotating.

ROTLIST does not make a copy of the list before rotating the items in the list. The list is modified in place. To keep a copy of the original list, use COPYLIST before ROTLIST.

An error condition results if the list has the NOUPDATE attribute.

To check a list's attributes, use HASATTR. To change these attributes, use SETLATTR.

Example

Manipulate the list identified by LISTID, which contains the five character values A, B, C, D, and E. Display the list, rotate it right to left one time and display that list, and then rotate it left to right twice and display that list.

```
call putlist(listid,'Input list=');
listid = rotlist(listid); /* Rotate 1 time */
call putlist(listid,'Rotated  1=');
```

```
listid = rotlist(listid,-2);
call putlist(listid,'Rotated -2=');
```

The preceding statements produce the following changes. The net result is that the list is rotated backwards one time.

```
Input list=('A' 'B' 'C' 'D' 'E')[3]
Rotated  1=('B' 'C' 'D' 'E' 'A')[3]
Rotated -2=('E' 'A' 'B' 'C' 'D')[3]
```

Note: [3] is the list identifier that was assigned when this example was run and may be different each time the example is run. △

See Also

"HASATTR" on page 458

"REVLIST" on page 617

"SETLATTR" on page 656

"SORTLIST" on page 669

RSESSION

Returns the name, description, and SAS software version of a remote session

Category: Interface to SAS Software

Requires SAS/CONNECT software

Syntax

cval=**RSESSION**(*n*);

cval
contains up to 48 characters of information identifying a remote session. Characters 1 through 8 contain the session identifier (the REMOTE= value). Characters 9 through 48 contain the description.
Type: Character

n
is the number of the remote session to identify.
Type: Numeric

Details RSESSION returns the session identifier and the corresponding description for a remote session that has been established with SAS/CONNECT software. You must have previously defined the description using RSTITLE.

If no remote link exists, the returned value is blank. If a link exists but no description has been specified, then characters 9 through 48 in the returned value are blanks.

See *SAS/CONNECT User's Guide* for more information about establishing a link between local and remote hosts.

Example

Retrieve the name and description of remote session number 1:

```
value=rsession(1);
```

See Also

"RLINK" on page 621

"RSTITLE" on page 624

RSTITLE

Defines a description for an existing connection to a remote session

Category: Interface to SAS Software

Requires SAS/CONNECT software

Syntax

sysrc=**RSTITLE**(*session-id*,*description*);

sysrc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

session-id
is one to eight characters that identify the remote session (the REMOTE= value).
Type: Character

description
is one to 40 characters to associate with the remote session.
Type: Character

Details You can retrieve the information that RSTITLE saves by using RSESSION to build a list of connections. You can then use the list to select a connection when submitting statements to a remote host.

In order to use this function, Release 6.07 or later of SAS software must be running on both the local and remote host systems.

See *SAS/CONNECT User's Guide* for more information about establishing a link between local and remote hosts.

Example

Define the description **`MVS Payroll Data`** for the remote session that has the identifier **`A`**:

```
session='A';
   description='MVS Payroll Data';
```

```
rc=rstitle(session,description);
```

See Also

"RLINK" on page 621

"RSESSION" on page 623

RUN

Stops executing statements in the program section that is currently executing

Category: Control Flow

Comparisons: SAS Statement with limitations in SCL

Alias: RETURN

Syntax

RUN;

Details In SCL, RUN is treated as an alias for RETURN.

For details about the RUN statement in the base SAS language, see *SAS Language Reference: Dictionary*.

See Also

"RETURN" on page 615

"STOP" on page 674

SASNAME

Verifies that a name is a valid SAS name

Category: Utility

Syntax

rc=**SASNAME**(*name*);

rc

contains the return code for the operation:

1	The name is a valid SAS name.
0	The name is not a valid SAS name.

Type: Numeric

name
is the name to be verified as a valid SAS name.
Type: Character

Details SASNAME verifies that a specified name is a valid SAS name. SAS names can be up to 32 characters long. The first character must be a letter (A, B, C, . . . , Z) or underscore (_). Other characters can be letters, numbers (0, 1, . . . , 9), or underscores. Blanks cannot appear in SAS names, and special characters (for example, $, @, #), except underscores, are not allowed.

Example

```
erroroff catalogname;
rc=sasname('catalogname');
if (rc=0) then do;
   erroron 'catalogname';
   _msg_= 'Catalog name is invalid.';
   end;
```

Note: In this example, the value for CATALOGNAME must be a one-level SAS name. SASNAME considers a two-level name of the form *libref.catalog-name* invalid because it contains the dot (.) character. △

SASTASK

Determines whether a SAS procedure is running

Category: Interface to SAS Software

Syntax

rc=**SASTASK**();

rc
contains the return code for the operation:

1	A SAS procedure is active.
0	No SAS procedure is active.

Type: Numeric

Example

Determine whether a SAS procedure is currently running before attempting to submit code to SAS software. If so, display a message to inform the user why the code cannot be submitted.

```
if (sastask()) then
 _msg_='Another procedure is currently active.';
else
   do;
      submit continue;
```

```
            data a;
               x=1;
            run;
         endsubmit;
      end;
```

SAVEENTRYDIALOG

Opens a dialog window that lists catalog entries, and returns the name of the selected entry

Category: Selection List

Syntax

selection=**SAVEENTRYDIALOG**(<*type*<,*initial*<,*entry-name*<,*list-id*<,*description*>>>>>);

selection
contains the four-level name of the selected SAS file, or a blank if nothing is selected.
Type: Character

type
specifies member types to list in the dialog window, such as CLASS, SCL, or FRAME. This can reduce the length of the list in the dialog window. If *type* is not used, names of all catalog entry types in the data library are listed.
Type: Character

initial
is the four-level name of the catalog entry to be the initially selected item in the dialog window when it opens.
Type: Character

entry-name
is a two- or four-level name of a catalog entry to call when a user selects the Advanced button in the dialog window. The entry can be any of the following types: FRAME, SCL, PROGRAM, HELP, CBT, or MENU.
If *entry-name* is not specified, then the window does not contain Advanced.
Type: Character

list-id
contains the identifier of an SCL list that is passed to *entry-name*.
Type: Numeric or List

description
contains the description of the returned *selection*.

Note: This parameter is an update parameter. See “Input, Output, and Update Parameters” on page 34 for more information. △
Type: Character

Details SAVEENTRYDIALOG enables you to implement a **`Save As`** choice by displaying a dialog window that lists entries in SAS catalogs. SAVEENTRYDIALOG returns a user’s selection, which enables you to create code that performs the save action. The entry can be saved under a different name or the same name.

If *entry-name* is supplied, the item selected in the dialog window is passed to *entry-name* through the following variables, which are created automatically:

USERDATA
: contains the list passed in the call to the OPENENTRYDIALOG and SAVEENTRYDIALOG functions.

LIBREF
: contains the libref of the selected item.

CATALOG
: contains the catalog name of the selected item.

ENTRY
: contains the entry name of the selected item.

TYPE
: contains the member type of the selected item.

List-id enables information to be passed between the calling entry and *entry-name*. It is passed to *entry-name* through the USERDATA variable.

Examples

Example 1: Saving a Catalog Entry Select a catalog entry of type FRAME to save and display the selected entry.

```
selection=saveentrydialog('frame');
call display(selection);
```

Example 2: Using the Advanced Button with an SCL List Specify an entry to call when a user selects Advanced, as well as an SCL list that contains values to pass to the entry.

```
dcl list mydata;
mydata=makelist();
rc=insertc(mydata,'test');
selection=saveentrydialog('frame','mylib.mycat.inital.frame',
   'mylib.mycat.myentry.frame',mydata, description);
if sysrc(1)=-1 then do;
   ...SCL statements to execute when
   the user cancels from the window...
   rc=dellist(mydata);
end;
else do;
   ...SCL statements to handle selections...
end;
```

The SCL entry for MYLIB.MYCAT.MYENTRY.FRAME contains the following program:

```
dcl char(8)  libref;
dcl char(8)  type;
dcl char(32) catalog;
dcl char(32) entry
dcl list userdata;

init:
put libref=;
put catalog=;
put entry=;
put type=;
call putlist(userdata);
return;
```

Note: The SCL entry must declare these exact variables, in the order specified, to properly reference the selected entry. △

If the user selects MYLIB2.MYCAT2.MYENTRY2.FRAME in the dialog window and then selects Advanced, the output for this program is

```
libref=Mylib2
catalog=Mycat2
entry=Myentry2
type=Frame
('test')[1]
```

SAVELIST

Stores SCL list items in a SAS catalog entry or in an external file

Category: List

Syntax

sysrc=**SAVELIST**(*type*,*target*,*list-id*<,*attr-list-id*<,*description*>>);

sysrc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

type
specifies the type of file or data source named in *source* and one or more options to use:

'CATALOG<(*options*)>**'**
saves text in a SAS catalog entry.

'FILE<(*options*)>**'**
saves text in an external file that is named in *target*.

'FILEREF<(*options*)>**'**
saves text in an external file that is identified by a fileref named in *target*.
Type: Character
The available options are described in "*Type* Options" on page 630. Separate multiple options with blanks. For example, when you save a list to a catalog entry, the first character of each item is assumed to be a carriage-control character, because FILLIST puts a carriage control in the first character of each item when it fills a list. If a list was not filled with FILLIST and does not have a carriage-control character as the first character, the first character of text is lost in each item of the list saved with SAVELIST. For lists of this type, use CATALOG(ADDCC) to add a carriage-control character as the first character in each list item.

target
is the name of the catalog entry, external file, or fileref in which the list items are stored. For catalog entries, this must be a four-level name (*libref.catalog.entry-name.entry-type*).
Type: Character

list-id
contains the identifier of the list that contains the items to be stored in a SAS file or external file. An invalid *list-id* produces an error condition. For text catalog entries, the first character in each item in the list contains the FORTRAN carriage-control character: 1 means that a new page starts with this line.
Type: Numeric or List

attr-list-id
contains the identifier of the list to fill with text attribute information when *type* is **CATALOG**. An error condition results if *attr-list-id* is not a valid list identifier.
Type: Numeric

description
is the text of a catalog entry description. This argument is ignored if the value for *type* is **FILE** or **FILEREF**. (The description is displayed in the catalog directory.)
Type: Character

Type Options

ADDCC
adds a default carriage-control character. Used with *type* **FILE**, **FILEREF**, and **CATALOG** and with catalog entry types **LOG**, **OUTPUT**, and **SOURCE**. If you use **ADDCC** and **STRIPCC** together, then both options are ignored.

APPEND
attempts to open the external file in APPEND mode and appends text from the list to the external file. Used with *type* **FILE** or **FILEREF**.

PRINT
designates an external file as a PRINT file (uses host carriage-control characters). Used with *type* **FILE** or **FILEREF**.

STRIPCC
removes carriage-control characters. Used with *type* **FILE**, **FILEREF**, and **CATALOG** and with catalog entry types **LOG**, **OUTPUT**, and **SOURCE**. If you specify **STRIPCC**, then carriage-control characters are ignored and default carriage control is used. If you use both **STRIPCC** and **ADDCC**, then both options are ignored.

TRIM
trims trailing blanks. Used with *type* **FILE**, **FILEREF**, and **CATALOG** and with catalog entry types **LOG**, **OUTPUT**, and **SOURCE**.

Details SAVELIST stores the items from an SCL list into a SAS catalog entry or an external file.

When *type* is **CATALOG** and you specify LOG, SOURCE, or OUTPUT as the *entry-type* of *target*, SAVELIST assumes that the first character of each item is a carriage-control character, because the default behavior of FILLIST for these entry types is to put a carriage-control character in the first item of list items when it creates a list. Therefore, if the items in the list identified by *list-id* do not have a carriage-control character as the first character, then save the list to a SOURCE entry using the ADDCC option. That will add a default (`' '`)carriage-control character as the first character in each line of text that is written to the SOURCE entry.

When *type* is **CATALOG** and you specify SLIST as the *entry-type* of *target*, the list — including names, list attributes, and item attributes — can be re-created with the FILLIST function, although the list identifiers will be different. The lists that you save with SAVELIST can persist across SAS sessions if you save them in a permanent catalog.

When a list is stored into any file type other than an SLIST entry, each item in the list identified by *list-id* must be a character string. Each string is stored as a separate line of text. When *type* is **CATALOG** and the entry type of *target* is LOG, OUTPUT, or SOURCE and a value is specified for *attr-list-id*, the attribute list items must also contain text. See the description of FILLIST for a description of the text attribute specifications. If the value for *attr-list-id* is omitted or is 0, then no attributes are stored with the catalog entry. Any value specified for *attr-list-id* is ignored when a list is stored in an external file or in an SLIST catalog entry.

When SAVELIST writes a list, an item that has the NOWRITE attribute is not written to the file. This is useful for placing temporary run-time values into a list that should not be written to the file because of its transient nature. For example, if you place table identifiers in lists to be saved with SAVELIST and restored with FILLIST in another task or another SAS session, the table identifiers become invalid. Thus, use SETLATTR to set the NOWRITE attribute on that list item.

(Do not confuse text attributes such as color and highlight with list attributes as specified with SETLATTR.) To check the attributes of a list or list item, use HASATTR. To change attributes, use SETLATTR.

Example

Perform operations similar to copying a list recursively with COPYLIST(mylistid,'Y'):

```
   /* Assume that the catalog WORK.TEMP exists: */
rc=savelist('catalog','work.temp.mylist.slist',
   mylistid);
newlistid=makelist();
rc=fillist('catalog','work.temp.mylist.slist',
   newlistid);
rc=delete('work.temp.mylist.slist','catalog');
```

See Also

"FILLIST" on page 407

SAVESASFILEDIALOG

Displays a dialog window that lists SAS files, and returns the name of the selected file

Category: Selection List

Syntax

selection=**SAVESASFILEDIALOG**(<*type*<,*level–count*<,*initial*<,*entry-name*<,*list-id*>>>>>);

selection
contains the two- or three-level name of the selected SAS file, or a blank if nothing is selected.
Type: Character

type
specifies a member type to list in the dialog window, such as DATA, VIEW, MDDB, and CATALOG. This can reduce the length of the list in the dialog window. If *type* is not used, the names of all SAS files in the data library are listed.
Type: Character

level-count
specifies whether the function returns a two- or three-level name. The only valid choices are 2 and 3. The default is two.
Type: Numeric

initial
is the name of the SAS file to be the initially selected item in the dialog window when it opens.

entry-name
is a two- or four-level name of a catalog entry that is called when a user selects the $\boxed{\text{Advanced}}$ button in the dialog window. The entry can be any of the following types: FRAME, SCL, PROGRAM, HELP, CBT, or MENU.
If *entry-name* is not specified, then the window does not contain $\boxed{\text{Advanced}}$.

list-id
contains the identifier for an SCL list that you can pass when *entry-name* is specified. The items in this list are the items to be displayed in the application's Advanced window for selection by application users.

Details SAVESASFILEDIALOG enables you to implement a **`Save As`** choice by displaying a dialog window that lists SAS files. SAVESASFILEDIALOG returns a user's selection, which enables you to create code that performs the save action. The SAS file can be saved under a different name or the same name.

If *entry-name* is supplied and $\boxed{\text{Advanced}}$ is selected, the item selected in the dialog window is passed to *entry-name* through the following variables:

USERDATA
contains the list that is passed in the call to SAVESASFILEDIALOG.

LIBREF
contains the libref of the selected item.

MEMBER
contains the member name of the selected item.

TYPE
contains the member type of the selected item.

List-id enables information to be passed between the calling entry and *entry-name*. It is passed to *entry-name* through the USERDATA variable.

Examples

Example 1: Saving a SAS File Select a SAS file of type DATA to save and display the table.

```
selection=savesasfieldialog('data view',3,'work.one');
```

Example 2: Using the Advanced Button with an SCL List Specify an entry of type DATA to be called when a user selects the Advanced button, as well as an SCL list that contains values to pass to the entry. The entry can be any of the following types: FRAME, SCL, PROGRAM, HELP, CBT, or MENU. If *entry-name* is not specified, then the window does not contain Advanced.

```
dcl list mydata;
mydata=makelist();
rc=insertc(mydata,'test');
selection = savesasfiledialog('data', 2','mylib.mycat.initial.frame',
   'mylib.mycat.myentry.frame',mydata);
if (sysrc(1)=-1) then do;
   ...SCL statements to handle when the user
   cancels from window...
end;
else do;
   ...SCL statements to handle selections...
end;
```

The SCL entry for MYLIB.MYCAT.MYENTRY.FRAME contains the following program:

```
dcl char(8) libref;
dcl char(32) member;
dcl char(8) type;
dcl list userdata;

init:
put libref=;
put member=;
put type=;
call putlist(userdata);
return;
```

Note: The SCL entry must declare these exact variables, in the order specified, to properly reference the selected entry. △

If the user selects MYLIB2.MYMEMBER2.DATA in the dialog window and then selects Advanced, the output for this program is

```
libref=Mylib2
member=Mymember2
type=Data
('test')[1]
```

Example 3: Using the LEVEL Parameter Select a SAS file to save and return different levels.

```
selection=savesasfiledialog ('data'',2);
put selection=
selection=savesasfiledialog('data',3);
put selection=
```

If the user selects MYLIB.MYMEMBER.DATA for both selections, the output would be

```
selection=mylib.mymember
selection=mylib.mymember.data
```

See Also

"OPENSASFILEDIALOG" on page 580

SAVESCREEN

Saves the values of data entry fields without exiting from the window

Category: Window

Syntax

CALL SAVESCREEN();

Details SAVESCREEN copies the current values of all window variables in a PROGRAM entry into the user's profile for later restoration. The values are stored for recall across invocations of SAS software in a catalog entry named SASUSER.PROFILE.*entry*.AFPGM, where *entry* is the name of the SAS/AF entry whose values are saved. This is similar to the SAVE command in the SAS/AF program window.

The saved values can be reloaded either with the RECALL command in the application window, or with the AUTORECALL=YES option in the AF command that invokes the application.

SAVESCREEN is valid only for PROGRAM entries.

Example

Save the final contents of the fields in an application window:

```
TERM:
   call savescreen();
return;
```

SCREENNAME

Returns the name of the current window

Category: Window

Syntax

name=**SCREENNAME**();

name
contains the four-level name of the current window.
Type: Character

Details The SCREENNAME function returns the name of the current window. For example, assume that there are two PROGRAM entries named SURVEY and NEWMAP in a catalog named MYLIB.TESTS. When SURVEY is executing, SCREENNAME returns MYLIB.TESTS.SURVEY.PROGRAM. When NEWMAP is executing, SCREENNAME returns MYLIB.TESTS.NEWMAP.PROGRAM.

In the FSEDIT and FSVIEW procedures, SCREENNAME returns the name of the SCREEN or FORMULA entry that the application is currently using.

Example

Display the window name on the message line:

```
_msg_=screenname();
```

SEARCH

Creates or manipulates the current catalog search path

Category: Catalog

Syntax

rc=**SEARCH**(*cat-name* | *special-argument*);

rc
contains the return code for the operation:

0	successful
–1	not successful

Type: Numeric

cat-name
is the catalog to push to the front of the current search list.
Type: Character

special-argument
are any of the following:

'-DISABLE'
to disable the current search list.

'-ENABLE'
to enable the current search list. If there is no disabled list, then the current search list remains empty.

'-POP'
to remove the first name from the current search list.

'-POPALL'
to clear the current search list.

Note: You must include the hyphen (-) as the first character in each of these argument values. Otherwise, the value will be treated as a catalog name in the WORK library. △

Type: Character

Details You do not need to know the exact location of a catalog member, only that it is in one of the catalogs in the current search path.

When a function that uses the current search path is called, the catalogs in the search path are searched from first to last until the specified entry is found or until the end of the search path is reached.

If there is no current search path, or if the current search path has been disabled or overridden, then the search is limited to the current catalog.

Example

Set up a search list with MYLIB1.CAT1, MYLIB1.CAT2 and MYLIB1.CAT3. Each time SEARCH is called, the new value of the argument is pushed to the front of the list. Therefore, MYLIB1.CAT3 is searched first. If MYPROG.PROGRAM exists only in MYLIB1.CAT1, it still executes correctly, because MYLIB1.CAT1 is also in the search path.

```
rc=search('mylib1.cat1');
rc=search('mylib1.cat2');
rc=search('mylib1.cat3');
call display('myprog.program');
```

See Also

"CEXIST" on page 276

"DISPLAY" on page 350

"SEARCHPATH" on page 638

SEARCHC, SEARCHL, SEARCHN, and SEARCHO

Search for a value in an SCL list and return its position number

Category: List

Syntax

index=**SEARCHC**(*list-id*,*cval*<,*start-index*
<,*occurrence*<,*ignore-case*<,*prefix*>>>>);

index=**SEARCHL**(*list-id*,*sublist-id*<,*occurrence*
<,*start-index*>>);

index=**SEARCHN***list-id*,*nval*(<,*occurrence*
<,*start-index*>>);

index=**SEARCHO**(*list-id*,*object-id*<,*occurrence*
<,*start-index*>>);

index
contains the index from the SCL list of the item that has the specified character value, or 0 if the value was not found.
Type: Numeric

list-id
is the identifier of the list to search. An invalid *list-id* produces an error condition.
Type: Numeric or List

cval
is the character value for SEARCHC to search for. *Cval* is compared only to the character values in the list.
Type: Character

sublist-id
contains the identifier of the sublist for SEARCHL to search for. *Sublist-id* is compared only to the list identifiers in the list.
Type: Numeric

nval
is the numeric value for SEARCHN to search for. *Nval* is compared only to numeric values in the list.
Type: Numeric

object-id
contains the identifier of the object for SEARCHO to search for. *Object-id* is compared only to the object identifiers in the list.
Type: Numeric or Object

occurrence
is the occurrence of the value to search for. The default, 1, indicates the first occurrence of the item.
Type: Numeric

start-index
is the position in the list at which to start the search for the item. By default, *start-index* is 1 (the first item). If *start-index* is positive, then the search begins at position *start-index* items from the beginning of the list. If *start-index* is negative, then the search begins at the item specified by ABS(*start-index*) items from the end of the list. An error condition results if the absolute value of *start-index* is zero or if it is greater than the number of items in the list.
Type: Numeric

ignore-case
specifies how SEARCHC should compare string values:

'Y'	ignores the case of the character strings.
'N'	does a case-sensitive comparison of the character strings. (This is the default.)

Type: Character

prefix
specifies whether the value should be treated as a prefix:

'Y'	does a prefix comparison and searches for any items that have *cval* as a prefix. SEARCHC compares only the first *m* characters, where *m* is the length of *cval*.
'N'	does not do a prefix search but compares all characters to *cval*. (This is the default.)

Type: Character

Details SEARCHC, SEARCHL, SEARCHN, and SEARCHO do not search for a value in any sublists of the list identified by *list-id*.

If *occurrence* and *start-index* are both positive or both negative, then the search proceeds forward from the *start-index* item. For forward searches, the search continues only to the end of the list and does not wrap back to the front of the list. If either *occurrence* or *start-index* is negative, then the search proceeds from the last item toward the beginning of the list. For backward searches, the search continues only to the beginning of the list and does not wrap back to the end of the list.

To search for an item by name rather than by value, use NAMEDITEM.

Examples

Example 1: Using the SEARCHC Function Find the position of the next-to-last occurrence of a string that begins with **SAS**, ignoring case:

```
last2=searchc(mylistid,'sas',2,-1,'Y','Y');
```

Example 2: Using the SEARCHL Function Search the list identified by MYLISTID for the third occurrence of the identifier for the sublist item identified by the value of NAMELISTID:

```
third=searchl(mylistid,namelistid,3);
```

Example 3: Using the SEARCHN Function Search for the third occurrence of the number 46 in the list identified by MYLISTID:

```
third=searchn(mylistid,46,3);
```

Example 4: Using the SEARCHO Function Search the list identified by MYLISTID for the third occurrence of the identifier for the object BUTTON:

```
third=searcho(mylistid,objectid,3);
```

See Also

"NAMEDITEM" on page 555

SEARCHPATH

Reports the complete pathname of a SAS catalog entry

Category: Catalog

Syntax

path-name=**SEARCHPATH**(*entry-name*<,*description*>);

path-name
contains the path (four-level name) for the specified entry if it was found in the current catalog or in the search path. If the entry was not found, then the value is blank.
Type: Character

entry-name
is the SAS catalog entry to search for.
Type: Character

description
specifies whether to return the description of the entry and the date the entry was last updated. Specify **Y** to return the description and the date.
Type: Character

Details SEARCHPATH returns the full four-level SAS library member name for a catalog entry if the entry is found in one of the catalogs that are in the current search path. It also returns the description that is stored with the entry in *description*.

If an entry that has the specified name appears in more than one catalog in the search path, then only the path to the first entry found is returned. If no search path is defined, the search is limited to the current catalog (the catalog in which the executing entry is stored).

To define the search path for your application, use SEARCH.

Note: To get the pathname and description of an entry, you can use the method interface of the Catalog entry class. △

Example

Load a stored list from a catalog entry if the entry is found in the current search path:

```
init:
   DCL num desclen
   rc=rc;
   rc=search('sashelp.afclass');
   rc=search('sashelp.aftools');
   rc=search<'sashelp.assist');

   path=searchpath('listed.frame','y');
   pathlen=length(path);
   name=scan(path,1,'');
   namelen=length(name) + 1;
   desclen = pathlen - (namelen) - 10;
   desc = substr(path, namelen + 1, desclen);
   date = substr(path, pathlen - 9, 10);
   put name=;
   put desc=;
   put date=;
return;
```

This program produces the following output:

```
name=sashelp.aftools.LISTED.FRAME
desc=   Generic list editor
date=10/09/1996
```

See Also

"SEARCH" on page 635

SELECT

Selects a specified row of a selection list

Category: Extended Table

Syntax

rc=**SELECT**(*row*);

rc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

row
is the number of the row to select.
Type: Numeric

Details The selection highlights the specified row. SELECT is useful for forcing the selection of a row. For example, you can use this function to set initial values or default values. Ordinarily, a user selects a row by pressing ENTER or clicking on the row with the mouse.

You can use SELECT only for selection lists that were built with extended tables in PROGRAM entries. FRAME entry controls must use the _selectRow method. Because you can define extended tables only in SAS/AF software, you cannot use SELECT in FSEDIT or FSVIEW programs.

In order for an extended table to be considered a selection list, you must specify a number of selections in the SETROW routine.

Example

Select row 5 of the selection list:

```
INIT:
   call setrow(10,2);
   rc=select(5);
return;
```

See Also

“ISSEL” on page 495

“NSELECT” on page 573

“SELECTED” on page 642

“SETROW” on page 664

“UNSELECT” on page 696

SELECT

Executes one of several statements or groups of statements

Category: Control Flow

Comparisons: SAS Statement with limitations in SCL

Syntax

SELECT<(*select-expression*)>;
  WHEN-*1* <(*when-expression*)>*statement(s)*;
  <. . .**WHEN**-*n* <(*when-expression*)>*statement(s)*;>
  <**OTHERWISE** <*statement(s)*>;>
END;

select-expression
is an expression that evaluates to a single value. This argument is optional. If used, *select-expression* must be in parentheses.
Type: Character

when-expression
is a constant or an expression that evaluates to a single value.
Type: Character

statement(s)
are one or more executable SAS statements, including DO, SELECT, and null statements. When used in a WHEN statement, a null statement causes SAS to recognize a condition as true without taking further action. In OTHERWISE statements, null statements prevent SAS from issuing an error message when all WHEN conditions are false.
Type: Character

Differences in SELECT Statement Execution

For SELECT groups in SCL, WHEN statements of the form WHEN(*a1*, *a2*, *a3*) are not supported. However, the following forms are supported:

- WHEN(*constant*)
- WHEN(*expression*).

OTHERWISE is an optional statement. If OTHERWISE is omitted, and if no WHEN conditions are met, the program halts.

Each WHEN statement implies a DO group of all statements until the next WHEN or OTHERWISE statement. Therefore, the following program is valid:

```
select(x);
   when(1)  call display('a');
     ...optionally, more SCL statements...
   when(2)  call display('b');
     ...optionally, more SCL statements...
   otherwise call display('bad');
     ...optionally, more SCL statements...
end;
```

For details about the SELECT statement in the base SAS language, see *SAS Language Reference: Dictionary.*

Example

This example shows how to use expressions with the SELECT statement:

```
select;
    when(x=1)  put 'one';
    when(2<x<5) put 'between two and five';
    when(x>5 or x<0) put 'other';
end;
```

SELECTED

Returns the number of the row that corresponds to a user's choice in a selection list

Category: Extended Table

Syntax

row=**SELECTED**(*nval*);

row
contains the number of the selected row, or –1 if the value specified for *nval* is greater than the total number of selections.
Type: Numeric

nval
is the number of the selection.
Type: Numeric

Details You can use SELECTED only for selection lists that were built with extended tables in PROGRAM entries. FRAME entry controls must use the _selected method. Because you can define extended tables only in SAS/AF software, you cannot use SELECTED in FSEDIT or FSVIEW programs.

In order for an extended table to be considered a selection list, you must specify a number of selections in the SETROW routine.

Example

Suppose the application displays a selection list that contains ten rows and that the user selects first row 3 and then row 5. If SELECTED is called with the value 2 specified for *nval* (as in the following statement), then the value returned in the variable ROW is 5, because row 5 was the second selection.

```
row=selected(2);
```

See Also

"ISSEL" on page 495

"NSELECT" on page 573

"SELECT" on page 639

"SETROW" on page 664

"UNSELECT" on page 696

SELECTICON

Displays a dialog window that contains a list of icons, and returns the value of the selected icon

Category: Selection List

Syntax

selected-icon=**SELECTICON**(<*initial-icon*>);

selected-icon
contains the value of the icon that is selected.
Type: Numeric

initial-icon
is the value of the icon that is active when the selector window opens. If *initial-icon* is not supplied, or if *initial-icon* is not a valid icon, then no icon is active when the selector window opens.
Type: Numeric

Details SELECTICON enables a user to select a SAS icon from a selection list.

SELF

Contains the identifier of the control for the currently executing method, or the identifier of the FRAME entry if the FRAME entry is not running as a method

Category: System Variable

Details _SELF_ is a system variable that is provided automatically by the FRAME entry in SAS/AF. The SCL compiler automatically creates a space for it in the SCL data vector.

SELF has a valid value only when the FRAME entry's SCL code is running or when a method is running.

See "USECLASS" on page 698 for information on bypassing _SELF_ references in a USECLASS block.

Example

Suppose a FRAME entry contains an icon. The icon's _select method is defined as follows:

```
SELECT:
   method;
      /* If the icon is a Version 8 icon,  */
      /* the following statement could be  */
      /*         _self_.icon=2;            */
   _self._setIcon(2);
endmethod;
```

When a user selects the icon, the _select method executes, and _SELF_ contains the identifier of the icon. In a FRAME entry, _SELF_ contains the identifier of the FRAME entry if the FRAME entry is not running as a method. For example, you can use _SELF_ to send a method to the FRAME entry from the INIT section.

```
INIT:
   _self_._setMsg_('in init section');
return;
```

See Also

"_METHOD_" on page 548

"SEND" on page 644

"SUPER" on page 681

"USECLASS" on page 698

"_VALUE_" on page 704

SEND

Sends a method to an object using its identifier and can return a value from a called method

Category: Object Oriented

Syntax

CALL SEND(*object-id*,*method-name*<,*parameters*>);

object-id
contains the identifier that is associated with the object for which the method is invoked.
Type: Numeric or Object

method-name
is the name of the method to send. The method must be defined for the object's class or for one of the classes from which the object inherits methods. Case and trailing blanks are ignored in method names.
Type: Character

parameters
specifies one or more numeric or character parameters that are required by the method. Use commas to separate multiple parameters.

Note: These parameters are update parameters. See "Input, Output, and Update Parameters" on page 34 for more information. △
Type: Numeric, Character

Details SEND passes one or more arguments to a method in the form of parameters to the routine. The method may modify any of these parameters and pass values back to the calling program via the parameters, or the method may modify the object's automatic system variables. You can use the _getWidget method to return the *object-id* for a control.

You can also use SEND as a function if the called method returns a value with a RETURN statement.

The classes provided with SAS/AF software include a set of predefined methods. Subclasses that you define from these classes inherit those methods. You can also define your own methods. Methods are defined with the METHOD statement in an SCL entry, or they may be entire SAS/AF entries. (SCL, PROGRAM, FRAME, HELP, and MENU entries are allowed.) A METHOD statement uses the syntax of an ENTRY statement to declare the types and names of the parameters that the method expects.

The parameters that are passed to SEND must match the parameter definitions of the METHOD or ENTRY statement of the method. You can specify optional parameters, using the OPTIONAL= option for the METHOD or ENTRY statement of the method. You can specify variable lengths and types for parameters, using the ARGLIST= and REST= options in the METHOD or ENTRY statement of the called method.

The same method may be defined for one or more classes; each class has its own definition of the method. Therefore, when a method is invoked, the appropriate method definition is determined based on the object's class. If the specified method is not defined for the object's class, SAS/AF searches the hierarchy of parent classes for the method definition.

When a method executes, the SCL variable _SELF_ is automatically initialized to the object identifier *object-id*, enabling the method to invoke other methods for the same object. Also, any of the object's automatic system variables are initialized if the SCL program uses a variable of the same name and type as the automatic system variable. If a character variable named _METHOD_ is declared, it will be initialized with the method name.

If an SCL method executes a SEND or otherwise invokes a method, the values of all automatic SCL variables in the calling method are copied into the object. After the called method executes, the automatic SCL variables are re-initialized with the values of the caller's system variables. Other routines that execute methods are APPLY, NOTIFY, SUPAPPLY, and SUPER.

To send methods to Version 8 objects, you should use dot notation instead of CALL SEND. Dot notation provides compiler time checking and better performance at run time. For more information about dot notation, see "Accessing Object Attributes and Methods With Dot Notation" on page 119.

Using dot notation is the only way to call overloaded methods, because the compiler must be able to check method signatures in order to call the correct method.

For example, to send a message to an object using dot notation, you could use

```
_frame_._setMsg('Table '||tablename||
     ' does not exist');
```

(The system variable _FRAME_ contains the identifier for the frame.)

Note: If a component is a control in an extended table, then you can invoke methods only during the getrow and putrow sequences or for _init and _term methods. Also, in a

FRAME SCL entry, to send methods to controls in an extended table, you can use NOTIFY rather than SEND. △

Example

Send a _term method to an icon whose name is ICON1 and whose identifier is stored in the variable ICON1ID:

```
call send(_frame_,'_getWidget','icon1',icon1id);
call send(icon1id,'_term');
```

See Also

"APPLY" on page 252

"ENTRY" on page 369

"INSTANCE" on page 485

"LOADCLASS" on page 525

"LOADRES" on page 526

"METHOD" on page 540

"NOTIFY" on page 572

"SUPAPPLY" on page 679

"SUPER" on page 681

SET

Links SAS table columns to SCL variables of the same name and data type

Category: SAS Table

Syntax

CALL SET(*table-id*);

table-id
contains the identifier that was assigned when the table was opened. If *table-id* is invalid, the program halts.
Type: Numeric

Details Using the SET routine can significantly reduce the coding required for accessing the values of variables for modification or verification. After a CALL SET, whenever a read operation is performed from the SAS table, the values of the SCL variables are set to the values of the corresponding SAS table columns. If the lengths do not match, then the values are truncated or padded as needed. When UPDATE or APPEND is called, the values that are written to the SAS table are the values of the SCL variables. If you do not use SET, then you must use GETVARC, GETVARN, PUTVARC, and PUTVARN to explicitly move values between table columns and SCL variables.

For each read/write operation that is performed on a SET SAS table, SCL loops through all the SAS table columns and updates the corresponding SCL variables. If the mapped SCL variables are only a small subset of the total SAS table columns, the looping could slow the process and prevent optimal performance. To enhance the performance of the application, you could open a SAS table as follows, using an option that limits the columns to only those to be set:

```
dsid=open('sasuser.employee (keep=age name)');
call set(dsid);
```

SET links only SCL variables that are accessible to the entire SCL program. SET does not link local variables, and using SET to link local variables in CLASS and USECLASS blocks may cause compilation errors.

As a general rule, use SET immediately following OPEN if you want to link table columns and SCL variables. Character variables that are associated with table columns must be declared with a DECLARE or LENGTH statement. Otherwise, the SCL compiler considers these variables to be numeric and thus sets them to missing instead of copying the appropriate character value from the table column.

If you use SET, do not use PUTVARN and PUTVARC for any variables that would be linked by SET. UPDATE and APPEND automatically move the data from the SCL data vector to the table data vector before writing the row to the physical file.

If a table column and a Version 8 frame control have the same name, and if the table column has the same data type as the frame control's default attribute, SET links the frame control's default attribute with the table column.

Example

Automatically set the values of the SCL variables NAME and SALARY when a row is fetched for a window that contains the fields NAME and SALARY. The SAS table PERSONEL has three columns: NAME, SALARY and DEPT.

```
tableid=open('personel','i');
call set(tableid);
rc=fetchobs(tableid,10);
```

See Also

"APPEND" on page 251

"FETCH" on page 389

"FETCHOBS" on page 390

"GETVARC and GETVARN" on page 450

"LOCATEC and LOCATEN" on page 527

"PUTVARC and PUTVARN" on page 609

"UPDATE" on page 697

SETCR

Controls the cursor's response to the carriage-return key

Category: Control or Field

Syntax

CALL SETCR(*advance*,*return*<,*modify*>);

advance
specifies how the cursor moves when a user presses the carriage-return key:

'STAY'
The cursor does not move.

'HTAB'
The cursor moves to the next field in the same row. This option makes the carriage-return key work like a horizontal tab key. When the last field in the current row is reached, the cursor moves to the first field in the next row.

'NEWL'
The cursor moves to the first field in the next line. This option makes the carriage-return key work like a new-line key. When the last line is reached, the cursor moves to the first field in the first line.

'VTAB'
The cursor moves to the first field in the next line in the current column. This option makes the carriage-return key work like a vertical tab key. When the last field in the current column is reached, the cursor moves to the top of the next column.

'HOME'
The cursor moves to the command line, or to the first field in the window if the window has no command line.
Type: Character

return
specifies whether a carriage return passes control back to the application:

'RETURN'
A carriage return passes control to the application, whether or not a field is modified. That is, the MAIN section of an SCL program is executed.

'NORETURN'
A carriage return does not pass control to the application unless a field is modified.
Type: Character

modify
specifies whether the field should be marked as modified:

'MODIFY'
A carriage return on a field is considered a modification of the field unless the field is protected.

'NOMODIFY'
A carriage return on a field is not considered a modification of the field.
Type: Character

Details SETCR works like a more powerful version of CONTROL ENTER for defining the behavior of the carriage-return key. This routine overrides the ENTER or NOENTER option of the CONTROL statement.

FRAME entries ignore SETCR.

Example

Move the cursor vertically to the first field in the next line in the current column when the user presses the carriage-return key. Control does not pass to the application, and the field is not modified by a carriage return.

```
call setcr('vtab','noreturn','nomodify');
```

See Also

"CONTROL" on page 302

SETFKEY

Assigns a command to a function key

Category: Keys

Syntax

CALL SETFKEY(*key-name,command*);

key-name
is the function key name, as listed in the KEYS window. Function key names vary depending on the device that is being used.
Type: Character

command
is the command to assign to the key.
Type: Character

Details You can use SETFKEY only in entries that have a DISPLAY window containing fields or text. You cannot use it to assign function key settings in windows that use BLOCK to display block menus.

Example

Use FKEYNAME to return the name of a particular function key and GETFKEY to return the command that is assigned to the function key. If the command is not CANCEL, then SETFKEY assigns the CANCEL command to the function key.

```
INIT:
   keyname=fkeyname(1);
   command=getfkey(keyname);
   if (command ne 'CANCEL') then
      call setfkey(keyname,'CANCEL');
return;
```

See Also

"FKEYNAME" on page 413

"GETFKEY" on page 440

"NUMFKEYS" on page 574

SETFLD

Assigns a value to up to ten blank variables

Category: Control or Field

Syntax

CALL SETFLD(*value,variable-1<, . . . ,variable-10>*);

value
is the character value to assign.
Type: Character

variable-1, . . . ,variable-10
are up to ten character variables whose values you want to change from blank to *value*.
Type: Character

Details If the variable is blank, *value* is assigned to the variable. No values are changed for variables that are not blank.

This function is useful for setting the default values for a series of fields.

Example

Set each of the variables SYM1 through SYM5 to the value **-REQUIRED-** for each variable that is blank:

```
call setfld ('-REQUIRED-',sym1,sym2,sym3,sym4,sym5);
```

The above statement is equivalent to the following statements:

```
if (sym1=' ') then sym1='-REQUIRED-';
if (sym2=' ') then sym2='-REQUIRED-';
if (sym3=' ') then sym3='-REQUIRED-';
if (sym4=' ') then sym4='-REQUIRED-';
if (sym5=' ') then sym5='-REQUIRED-';
```

See Also

"CLRFLD" on page 293

SETFOOT

Sets the text of a footnote definition

Category: SAS System Option

Syntax

CALL SETFOOT(*foot-num*,*foot-text*);

foot-num
is the number (1 to 10) of the footnote definition to create or modify.
Type: Numeric

foot-text
is the text for the footnote definition.
Type: Character

Details SETFOOT works just like the FOOTNOTE statement. It clears all footnote definitions that are numbered higher than the one created. You cannot use SETFOOT to set graphic options such as color, tint, and position.

You can view footnote definitions in the FOOTNOTES window by using the FOOTNOTE command. Changing any text in the FOOTNOTES window, however, resets all graphically defined footnote options such as color, font, and position.

For more information about footnotes, see *SAS Language Reference: Dictionary*. For more information about graphical footnotes, see *SAS/GRAPH Software: Reference*.

Example

Create a footnote numbered 5. The statement deletes all footnotes with numbers greater than 5.

```
call setfoot(5,'This is the Fifth Footnote');
```

See Also

"GETFOOT" on page 441

"GETTITLE" on page 449

"SETTITLE" on page 666

SETITEMC, SETITEML, SETITEMN, and SETITEMO

Store a value at an indexed position in an SCL list

Category: List

Syntax

rc=**SETITEMC**(*list-id*,*cval*<,*index*<,*autogrow*>>);

rc=**SETITEML**(*list-id*,*sublist-id*<,*index*<*autogrow*>>);

rc=**SETITEMN**(*list-id*,*nval*<,*index*<,*autogrow*>>);

rc=**SETITEMO**(*list-id*,*object-id*<,*index*<,*autogrow*>>);

rc
is the *list-id*, which is the identifier of the modified list.
Type: Numeric

list-id
is the identifier of the list into which the value is stored. An invalid *list-id* produces an error condition.
Type: Numeric or List

cval
is the character value for SETITEMC to store in the list.
Type: Character

sublist-id
is the identifier of the sublist for SETITEML to store in the list. An invalid *sublist-id* produces an error condition.

nval
is the numeric value for SETITEMN to store in the list.
Type: Numeric

object-id
is the identifier of the object for SETITEMO to store in the list. An invalid *object-id* produces an error condition.
Type: Numeric or Object

index
is the position of the item in the list. The position can be specified as a positive or negative number. By default, *index* is 1 (the first item). If *index* is a positive number, then the item is at position *index* from the beginning of the list. If *index* is a negative number, then the item is at position ABS(*index*) from the end of the list. An error condition results if the absolute value for *index* is zero or if it is greater than the number of items in the list.
Type: Numeric

autogrow
specifies whether the list can expand to accommodate a new item:

'N'	The size of the list cannot change. (This is the default.)
'Y'	The size of the list can increase to accommodate a new item that is being added to the list. The list expands only if *index* is greater than the current number of items in the list and the list does not have the FIXEDLENGTH attribute.

Type: Character

Details Using SETITEMC, SETITEML, SETITEMN, or SETITEMO is analogous to assigning a character, a sublist identifier, a numeric value, or an object identifier, respectively, to an indexed item in an array. *Index* specifies the position in the list of the item whose value is assigned. If *autogrow* is Y , then *index* can be greater than the length of the list. These functions then expand the list to a total of *index* items. They also set all other new items to missing values, and they place the new value into the list. These functions can add items only to the end of a list. Use INSERTC, INSERTL, INSERTN, or INSERTO to insert values elsewhere in a list.

SETITEMC, SETITEML, SETITEMN, and SETITEMO replace an existing item in a list and even change its type unless the item or the list has the FIXEDTYPE attribute.

SETITEMC, SETITEML, SETITEMN, and SETITEMO do not make a copy of the list before assigning the new item. The list is modified in place.

Note: The return value of these functions is not used to indicate whether an error has been detected. When an error occurs, the program simply halts. △

An error condition results

- □ if the absolute value of *index* is zero, or if it is greater than the number of items in the list and *autogrow* is N.
- □ if the absolute value of *index* is greater than the length of the list and the list has the FIXEDLENGTH attribute (even if *autogrow* is Y).
- □ if the list or item has the NOUPDATE attribute.
- □ if the list or item has the FIXEDTYPE attribute and the function attempts to set the item to a different type. For example, if item 4 is numeric and has the FIXEDTYPE attribute, then the following statement fails:

  ```
  list=setitemc(list,'abc',4);
  ```

- □ with SETITEMC or SETITEML, if the list has the NUMONLY attribute.
- □ with SETITEMN or SETITEML, if the list has the CHARONLY attribute.
- □ with SETITEML, if *sublist-id* identifies a local list and *list-id* identifies a global list. (You cannot place local lists into global lists.)

To check the attributes of a list or list item, use HASATTR. To change attributes, use SETLATTR.

See Also

"GETITEMC, GETITEML, GETITEMN, and GETITEMO" on page 442

"INSERTC, INSERTL, INSERTN, and INSERTO" on page 482

"NAMEITEM" on page 558

"SETNITEMC, SETNITEML, SETNITEMN, and SETNITEMO" on page 661

SETKEY

Defines an index key for retrieving rows from a SAS table

Category: SAS Table

Syntax

nval=**SETKEY**(*table-id*<,*key-name*<,*condition*<,*scroll-option*<,*list-id*>>>>);

nval

contains the return code for the operation:

0	An active key was successfully set or cleared.
≠0	An error or warning condition occurred.

Type: Numeric

table-id
is the identifier that was assigned when the table was opened. If *table-id* is invalid, the program halts.
Type: Numeric

key-name
is the index key to be used for retrieval.
Type: Character

condition
specifies comparison criteria for the key value:

'EQ'	equal to the key value (This is the default.)
'GE'	greater than or equal to the key value
'GT'	greater than the key value
'LE'	less than or equal to the key value
'LT'	less than the key value.

Type: Character

scroll-option
specifies whether rows can be randomly retrieved:

'SCROLL'
Rows can be retrieved in random order. (This is the default.)

'NOSCROLL'
Rows can only be retrieved sequentially. This option improves performance when the table is accessed via the REMOTE engine and the IS mode is specified for the second argument of the OPEN function. Those options reduce the number of data transfer operations that are required when the table is read.
Type: Character

list-id
is the identifier for the list that contains values for the index key variables. You must use SETNITEMC and SETNITEMN to assign the values to the corresponding key variables in the list. An invalid *list-id* produces an error condition.
Type: Numeric or List

Details SETKEY enables you to set an active key in an open table to a simple or composite key. It establishes a set of criteria for reading SAS table rows by comparing the value of the columns from the SDV to the key value in the rows.

Using a composite key with SETKEY operates the same way as the WHERE function only when the condition is 'EQ'. The value returned when the condition is 'EQ' is the same as if the columns specified in the composite key are connected by WHERE conditions using AND or ALSO.

For all other conditions (GT, GE, LT, or LE) specified with SETKEY for a composite key, the composite key columns are concatenated to form the index key. The number returned by the KEYCOUNT function is the number of rows in the table that satisfy the composite key. For example, if the composite index consists of columns SEX and AGE and the condition is GT (greater than), the values to search for are concatenated such that key values of F for SEX and 13 for AGE yield an index key of F13. Because the search is performed on the concatenated values, some values that you did not expect may meet a search condition. For example, key values of M for SEX and 11 for AGE meet the search condition, because the string M11 is considered greater than the string F13. If the active key is a composite key and the *condition* parameter is set to GT, GE, LT, or LE, the table subsets into rows whose primary key column values meet the specified criteria. Consequently, you still have to check the values of other key parts against the data vector to narrow down the subset of rows.

SETKEY works only after SET is called in the SCL program or when a list identifier is passed. The list identifier must point to a list that contains the values of the index key columns. Once an active key is set through SETKEY, it remains active until

- the table is closed
- another key is set
- the current setting is cleared by passing the table identifier alone to SETKEY.

The table is automatically positioned at the first row that meets the specified criteria. Use FETCH or FETCHOBS to read the row.

SETKEY returns an error code if a WHERE clause is in effect. Index keys cannot be used in conjunction with WHERE clauses.

Examples

Example 1: Defining an Index Key That Was Created Previously Define an index key for the table MYDATA, which subsets the table into only those rows where the value of the AGE column is greater than or equal to 20:

```
   /* Assuming a simple key, AGE, has been defined */
age=20;
dsid=open('MYDATA','I');
call set(dsid);
rc=setkey(dsid,'age','ge');
do while(fetch(dsid) ne -1);
   name=getvarc(dsid,1);
   put name=;
end;
```

Example 2: Using a Composite Index Key with GE Search the table CHILDREN for all boys who are 5 years old or older. The composite key ATTR, which is created by ICREATE, is used for retrieval. The values of the composite key columns are concatenated, and the search is performed on the combined value. In this example, the key selects rows where AGE||GENDER ≥ 5M. The FETCH function within the DO-loop returns all rows where AGE>=5. Because some of the rows may not have a matched concatenated key part, you need an additional check on the value of the GENDER column in order to skip unmatched rows.

```
dsid=open('children','v');
   /* Create a composite key ATTR with AGE */
   /* as primary key column               */
rc=icreate(dsid,'attr','age gender');
call set(dsid);
age=5;
gender='M';
rc=setkey(dsid,'attr','ge');
do while(rc=0);
      /* FETCH function applies the retrieval */
      /* criteria and retrieves all rows      */
      /* for which AGE >=5                    */
   rc=fetch(dsid);
   if (rc) then leave;
      /* Filter out rows with gender ne 'M' */
   if (upcase (gender) ne 'M') then continue;
```

```
      child=getvarc(dsid,varnum(dsid,'name'));
      put child=;
   end;
   rc = close (dsid);
```

Example 3: Using an SCL List Instead of CALL SET Using an SCL list avoids possible name collisions. Also, it enables you to set the retrieval criteria for rows at run time instead of at compile time.

```
   dsid = open ( 'children','v');
   rc = icreate( dsid, 'attr','age gender');
   list = makelist();
   list = setnitemc (list,cval,'gender');
      /* cval contains the value of 'M' */
   list = setnitemn (list,nval,'age');
      /* nval contains the value of 5 */
   rc = setkey (dsid,'attr','ge','',list);
      /* Print out all names with */
      /* age >= 5 and gender= 'M'    */
   do while ( rc= 0 );
   rc = fetch (dsid);
   if (rc) then leave;
      sex1 = getvarc (dsid, varnum(dsid, 'gender'));
      if (upcase (gender) ne 'M') then continue;
      child = getvarc (dsid, varnum(dsid, 'name'));
      put child=;
   end;
   rc = close (dsid);
```

See Also

"ICREATE" on page 464

"IDELETE" on page 468

"IOPTION" on page 490

"IVARLIST" on page 498

"KEYCOUNT" on page 499

"SET" on page 646

"WHERE" on page 718

SETLATTR

Sets the attributes of an SCL list or an item in a list

Category: List

Syntax

rc=**SETLATTR**(*list-id*,*attributes*<,*index*>);

rc

contains the return code for the operation:

0	successful
≠0	not successful

Type: Character

list-id

is the identifier of the list whose attributes or item attributes are set. An invalid *list-id* produces an error condition.

Type: Numeric or List

attributes

lists one or more attributes of the list or list item as shown in "Attribute Values for Lists and List Items" on page 657 and "Attribute Values for Lists Only" on page 658. Use blanks to separate multiple attributes. Attributes for lists are ignored when you are setting list item attributes, and attributes for list items are ignored when you are setting list attributes. This enables you to create a single attribute string that you can apply both to lists and to list items.

Type: Character

index

is the position of the list item whose attributes are being modified. The position can be specified as a positive or negative number. By default, *index* is 1 (the first item). If *index* is a positive number, then the item is at position *index* from the beginning of the list. If *index* is a negative number, then the item is at position ABS(*index*) from the end of the list. If *index* is zero or omitted, then SETLATTR sets list attributes. An error condition results if the absolute value for *index* is zero or if it is greater than the number of items in the list.

Type: Numeric

Attribute Values for Lists and List Items

'DEFAULT'

combines all the default attributes.

'DELETE'

allows the list or item to be deleted. (This is a default.)

'FIXEDTYPE'

prevents changes in the type of the item. See also NUMONLY and CHARONLY. For a list, prevents changes in the type of all individual items.

'NODELETE'

prevents a list or list item from being deleted. List items that do not have the NODELETE attribute can be deleted from a list with this attribute. A list without this attribute can be deleted even though it contains items that have the NODELETE attribute.

'NOFIXEDTYPE'

allows the type of an item to change. For a list, allows the type of each item to change as long as the list does not have the CHARONLY or NUMONLY attribute and the item does not have the FIXEDTYPE attribute. (This is a default.)

'NOUPDATE'

prevents updates to the value for a list item. For a list, updates are not allowed to any item, even those with the UPDATE attribute. This enables you to make a list read-only in one step without having to make each individual item read-only.

'UPDATE'
allows updates to the value of a list item. For a list, UPDATE allows updates to all items that do not have the NOUPDATE attribute. (This is a default.)

Attribute Values for Lists Only

'ANYNAMES'
allows item names to be any character string, although names are always converted to uppercase and trailing blanks are removed. (This is a default.)

'CHARONLY'
requires all items to have character values.

'COPY'
copies the list during a recursive copy operation if the list is a sublist. (This is a default.)

'DUPNAMES'
allows duplicate names in the list. (This is a default.)

'FIXEDLENGTH'
prevents the list length from changing.

'HONORCASE'
specifies that searches must be case-sensitive and that values for successful searches must match the case of the list item value.

'NOCHARONLY'
allows items to have numeric values and list identifier values. (This is a default.)

'NOCOPY'
prevents the list from being copied during a recursive copy operation if the list is a sublist. Instead, only the list identifier is copied to the target list. No recursion takes place for the list.

'NODUPNAMES'
requires all item names to be unique.

'NOFIXEDLENGTH'
allows the list length to change. (This is a default.)

'NOHONORCASE'
allows a search of list items to match values, regardless of the case in which they are stored. (This is a default.)

'NONUMONLY'
allows the list to contain character values and list identifier values. (This is a default.)

'NUMONLY'
requires all items to have numeric values.

'SASNAMES'
requires all items in the list to be named, and requires all names to be valid SAS names with no leading or trailing blanks.

Attribute Values for List Items Only

'ACTIVE'
makes the item active in a pop-up menu that is opened by the POPMENU function. (This is a default.)

'INACTIVE'
prevents the item from being active in a pop-up menu that is opened by the POPMENU function. (Users cannot select it.)

'AUTO'
specifies that the item is an automatic instance variable. Refer to *SAS/AF Software: FRAME Entry Usage and Reference, Version 6, First Edition* for more information on automatic instance variables.

'NOAUTO'
specifies that the item is not an automatic instance variable.

'NOWRITE'
prevents the item from being written by the SAVELIST function.

'WRITE'
allows the item to be written when the list is stored via SAVELIST. (This is a default.)

Details If *index* is omitted or zero, the attributes are assigned to the list. Otherwise, the attributes are assigned to the item at the position specified by *index*. Item attributes are attached to the items in the list, not to the position in the list, so that an item keeps its attributes after the list is rotated or reversed or after other items are inserted or deleted. That is, if you assign the NODELETE attribute to the fourth item in the list and then delete the item at position 2, the item still has the NODELETE attribute even though it is now at position 3.

If a list has the NOCOPY attribute, it is not copied if it is a sublist in a recursive call of COPYLIST. Instead, only the list identifier is copied to the target list, and no recursion takes place on the list. You can still copy a list with NOCOPY if it is the source (top-level) list.

Attribute Pairs

The attributes for lists and list items come in mutually exclusive pairs. For example, NOUPDATE designates that UPDATE is off. Setting one attribute of a mutually exclusive pair automatically turns off the other. The attribute pairs are shown in the following table.

Attribute	Complement Attribute	Applies To
ACTIVE	INACTIVE	Items
ANYNAMES	SASNAMES	Lists
COPY	NOCOPY	Lists
DELETE	NODELETE	Items and Lists
DUPNAMES	NODUPNAMES	Lists
HONORCASE	NOHONORCASE	Lists
NOCHARONLY	CHARONLY	Lists
NOFIXEDLENGTH	FIXEDLENGTH	Lists
NOFIXEDTYPE	FIXEDTYPE	Items and Lists
NONUMONLY	NUMONLY	Lists

Attribute	Complement Attribute	Applies To
UPDATE	NOUPDATE	Items and Lists
WRITE	NOWRITE	Items and Lists

The attributes of a list (such as CHARONLY, NUMONLY, FIXEDTYPE, and NOUPDATE) do not apply to sublists that the list contains. You must assign these attributes to the sublists if you want them to have these attributes.

Both list and item attributes are copied by COPYLIST. However, COPYLIST does not copy passwords.

Most of the list functions that alter lists or their contents are affected in some way by list attributes and item attributes. See the documentation for the individual functions to see how they are affected.

Using Passwords to Protect List Attributes from Modification

You can assign a password to list attributes, which enables you to protect them. The password is stored with the list and must be supplied in subsequent SETLATTR statements in order to modify either list attributes or item attributes. The password is specified as PASSWORD=*password* in the attribute string. The password may not contain any blanks, but case is significant. For example, the passwords frobble and FROBBLE are different.

The following statements show how to assign a password:

```
pwd='password=grombaq';
myattr='nodelete noupdate';
rc=setlattr(mylist,myattr||' '||pwd);
```

If an SCL program attempts to modify the value of one item in MYLIST, the program halts, because the list has the NOUPDATE attribute. If you want to permit updates of the list again, you can execute SETLATTR as follows:

```
rc=setlattr(mylist,pwd ||'update');
```

Note: An error condition results if you attempt to alter list attributes without specifying the PASSWORD= option and password. △

You can remove the password from a list with NOPASSWORD=. Thereafter, any SCL program can change attributes or set the password. Either of the following statements removes the previously set password from the list identified by MYLIST:

```
rc=setlattr(mylist,'no'||pwd);

rc=setlattr(mylist,'nopassword=grombaq');
```

You must supply the correct password in order to remove a password that was previously set.

Example

The NODELETE list attribute means that the list itself cannot be deleted with DELLIST. The NODELETE item attribute indicates that an item cannot be deleted from the list regardless of the item's type. The following statements show this distinction:

```
a=makelist(3);
b=makelist();
```

```
     /*  Set 3rd item in A to B.  */
  a=setiteml(a,b,3);
     /* Give list B the NODELETE attribute. */
     /* DELLIST(b) will be an error.            */
  rc=setlattr(b,'NODELETE');
     /* Give the 3rd item in A the      */
     /* NODELETE attribute.               */
     /* DELITEM(a, 3) will be an error.  */
  rc=setlattr(a,'NODELETE',3);
     /* Move B to the second item in the list */
     /* and set the third item with 0.        */
  a=setiteml(a,b,2);
  a=setitemn(a,0,3);
     /* Remove B from list A,           */
     /* but B still exists.             */
     /* DELITEM(a,2) will be an error now. */
  a=delitem(a,2);
```

See "CLEARLIST" on page 290 and "DELLIST" on page 336 for more information about the DELETE and NODELETE attributes.

See Also

"GETLATTR" on page 443

"HASATTR" on page 458

SETNITEMC, SETNITEML, SETNITEMN, and SETNITEMO

Assign a value to a named item in an SCL list

Category: List

Syntax

rc=**SETNITEMC**(*list-id*,*cval*,*name*,<*occurrence*
<,*start-index* <,*index*<,*forceup*>>>>);

rc=**SETNITEML**(*list-id*,*sublist-id*,*name*,<*occurrence*
<,*start-index*<,*index*<,*forceup*>>>>);

rc=**SETNITEMN**(*list-id*,*nval*,*name* ,<*occurrence*
<,*start-index*<,*index*<,*forceup*>>>>);

rc=**SETNITEMO**(*list-id*,*object-id*,*name*, <*occurrence*
<,*start-index*<,*index*<,*forceup*>>>>);

rc
is the *list-id*, which is the identifier of the modified list.
Type: Numeric

list-id
is the identifier of the list that contains the named item. An invalid *list-id* produces an error condition.
Type: Numeric or List

cval
is the character value for SETNITEMC to assign.
Type: Character

sublist-id
is the identifier of the sublist for SETNITEML to assign. An error condition results if *sublist-id* is not a valid identifier.
Type: Numeric

nval
is the numeric value for SETNITEMN to assign.
Type: Numeric

object-id
is the identifier of the object for SETNITEMO to assign. An error condition results if *object-id* is not a valid identifier.
Type: Numeric or Object

name
is the name of the item. If the named item is not found in the list, it is inserted into the list.
Type: Character

occurrence
specifies which occurrence of the named item to assign the specified value to, starting from the position specified in *start-index*. The default, 1, indicates the first occurrence of the item.
Type: Numeric

start-index
specifies where in the list to begin searching for the item. By default, *start-index* is 1 (the first item). If *start-index* is positive, then the search begins at position *start-index* items from the beginning of the list. If *start-index* is negative, then the search begins at the item specified by ABS(*start-index*) items from the end of the list. An error condition results if the absolute value of *start-index* is zero or if it is greater than the number of items in the list.
Type: Numeric

index
is a variable to contain the index of the modified or inserted item.

Note: This parameter is an update parameter. See "Input, Output, and Update Parameters" on page 34 for more information. △
Type: Numeric

forceup
specifies whether to conduct the name search in uppercase.

'N'	searches according to the list attributes HONORCASE and NOHONORCASE, which are specified with SETLATTR. (This is the default.)
'Y'	conducts the name search in uppercase regardless of the list attributes that are specified with SETLATTR. The HONORCASE and IGNORECASE attributes are ignored.

IGNORECASE is an alias for NOHONORCASE.
Type: Character

Details SETNITEMC, SETNITEML, SETNITEMN, and SETNITEMO do not make a copy of the list before modifying or inserting an item. The list is modified in place. If the named item is not found in the list (and if the list does not have the FIXEDLENGTH attribute), the item is inserted into the list.

If you specify a variable name for *index*, then these functions return the index number in the list of the modified or inserted item. You can reference this index to access the same item with these functions and other SCL list functions, as long as the items do not change positions (for example, as a result of an insert or delete operation). If the position of items in a list is stable, then using the *index* argument and subsequent index-based functions such as SETITEMC rather than name-based functions improves performance, because the list does not have to be searched multiple times to find a name match.

If *occurrence* and *start-index* are both positive or both negative, then the search proceeds forward from the *start-index* item. For forward searches, the search continues only to the end of the list and does not wrap back to the front of the list. If the named item is not found, it is inserted at the end of the list. If *occurrence* or *start-index* is negative, the search proceeds backwards from the end to the beginning of the list. For backward searches, the search continues only to the beginning of the list and does not wrap back to the end of the list. If the named item is not found, it is inserted at the beginning of the list.

The result of using SETNITEMC, SETNITEML, SETNITEMN, and SETNITEMO is similar to combining NAMEDITEM and SETITEM, SETITEML, SETITEMN, and SETITEMO, respectively. For example, consider the following statement:

```
mylist=setnitemc(mylist,'Jones','NAMES',1,1,i);
```

This statement performs the same operations as the following statements:

```
i=nameditem(mylist,'NAMES');
   /* If NAMES isn't found, insert it */
   /* at the end of the list. */
if i=0
then mylist=insertc(mylist,'Jones',-1,'NAMES');
else mylist=setitemc(mylist,'Jones',i);
```

Note: The return value of these functions is not used to indicate whether an error has been detected. When an error occurs, the program simply halts. △

An error condition results

□ if either the item or the list has the NOUPDATE attribute.

□ if either the item or the list has the FIXEDTYPE attribute and the new value is being assigned to an item that has a different type.

□ if the named item was not found and the list has the FIXEDLENGTH attribute.

□ with SETNITEMC and SETNITEML, if the list has the NUMONLY attribute.

□ with SETNITEMN and SETNITEML, if the list has the CHARONLY attribute.

□ with SETNITEML, if *list-id* identifies a global list and *sublist-id* identifies a local list. (Local lists cannot be placed into global lists.)

To check the attributes of a list or list item, use HASATTR. To change attributes, use SETLATTR.

See Also

"GETNITEMC, GETNITEML, GETNITEMN, and GETNITEMO" on page 445

"INSERTC, INSERTL, INSERTN, and INSERTO" on page 482

"NAMEDITEM" on page 555

"NAMEITEM" on page 558

"SETITEMC, SETITEML, SETITEMN, and SETITEMO" on page 651

"SETLATTR" on page 656

SETPARMID

Makes the value of an SCL numeric variable available between SCL programs

Category: Object Oriented

Syntax

CALL SETPARMID(*nval*);

nval
is the numeric value to store for retrieval by GETPARMID.
Type: Numeric

Details SETPARMID stores a number that can then be retrieved by calling GETPARMID. One program can use SETPARMID to store a value and another program can use GETPARMID to retrieve the value.

SETPARMID and GETPARMID allow only one value to be passed. To pass multiple values between entries, use the ENTRY statement. Other ways of making values available to other SCL programs include using macro variables and SCL lists.

Example

Open the SAS table MYDATA. Then use SETPARMID to store the table identifier value so that other programs can use GETPARMID to access the table:

```
dsid=open('mydata','i');
call setparmid(dsid);
```

See Also

"DISPLAY" on page 350

"ENTRY" on page 369

"GETPARMID" on page 448

SETROW

Determines the characteristics of extended tables

Category: Extended Table

Syntax

CALL SETROW(*num-rows*<,*num-sel*<,*sel-order*<,*dynamic*>>>);

num-rows
is the maximum number of rows for the table. If the table is dynamic, *num-rows* can be 0.
Type: Numeric

num-sel
is the number of items a user can select from the list. To display the list for information purposes only (no selections allowed), specify 0. To specify unlimited selections, use a value such as 9999 that is larger than the number of available selections. If *num-sel* is 1, then selecting one row deselects any previously selected row.
Type: Numeric

sel-order
specifies the selection order:

'Y'	Selected items are highlighted and are moved to the top of the list in the order in which they are selected.
'N'	Selected items are highlighted, but they are not moved to the top of the list. (This is the default.)
'A'	The selection list window automatically closes when the user makes a selection if only one selection is allowed. This option is valid only if *num-sel* is 1.
'B'	Combines Y and A.

Type: Character

dynamic
specifies whether the table is dynamic:

'Y'	specifies that the extended table is a dynamic table. Use the ENDTABLE routine in the getrow section to specify that no more rows are available.

Type: Character

Details In PROGRAM entries, a regular extended table has a specified number of rows, and a dynamic extended table has an unspecified number of rows. You cannot use the SETROW statement in the getrow or putrow section of an SCL program. In order to use SETROW, you must have specified the EXTENDED TABLE attribute from the GATTR window.

You can also use both regular and dynamic extended tables as selection lists. The second and third arguments, *num-sel* and *sel-order*, define a selection list. Making an extended table a selection list automatically turns on the block cursor. Use FIELD to turn off the block cursor.

To define a dynamic table that is not a selection list, specify 0 for *num-sel* and ''for *sel-order*.

You can use SETROW only on extended tables. Because you can define extended tables only in SAS/AF software, you cannot use SETROW in FSEDIT or FSVIEW programs. FRAME entries must use the _setMaxrow method.

Examples

Example 1: Specifying an Extended Table Specify an extended table that has 20 rows:

```
call setrow(20);
```

Example 2: Using an Extended Table as a Selection List Specify a selection list that has 20 rows. Three selections are allowed, and the selections are moved to the top of the table.

```
call setrow(20,3,'y');
```

Example 3: Specifying a Dynamic Extended Table Specify a dynamic table:

```
call setrow(0,0,'','y');
```

Example 4: Using a Dynamic Extended Table as a Selection List Specify a dynamic table to be used as a selection list. Three selections are allowed, and the selections are not moved to the top of the table.

```
call setrow(0,3,'n','y');
```

See Also

"CURTOP" on page 324

"ENDTABLE" on page 368

"TOPROW" on page 689

SETTITLE

Sets the text of a title definition

Category: SAS System Option

Syntax

CALL SETTITLE(*title-num*,*title-text*);

title-num
is the number (1 to 10) of the title definition to create or modify.
Type: Numeric

title-text
is the text for the title definition.
Type: Character

Details SETTITLE works just like the TITLE statement. It clears all title definitions that are numbered higher than the one created. You cannot use SETTITLE to set graphic options such as color, font, and position.

You can view title definitions in the TITLES window by using the TITLE command. However, changing any text in the TITLES window resets all graphically defined title options such as color, font, and position.

For more information about titles, see *SAS Language Reference: Dictionary*. For more information about graphical titles, see *SAS/GRAPH Software: Reference*.

Example

Create a title numbered 2. The statement deletes all titles with numbers greater than 2.

```
call settitle(2,'This is the Second Title');
```

See Also

"GETFOOT" on page 441

"GETTITLE" on page 449

"SETFOOT" on page 650

SHOWLIST

Displays a selection list window that contains up to 13 items, and returns the user's selections

Category: Selection List

Syntax

selection=**SHOWLIST**(*item-1*<, . . . ,*item-13*>,*message*);

selection
contains the selected items, or a blank if no selection is made. By default, *selection* is 200 bytes long. To accomodate values longer than 200 bytes, explicitly declare *selection* with a longer length.
Type: Character

item
lists up to 13 items, separated by commas, for the selection list.
Type: Character

message
is the text for a message to display above the selection list. Regardless of how many *item* values are supplied, the last argument is assumed to be the message. Use a null string (`''`)to specify the default message, which instructs users to make one selection.
Type: Character

Details SHOWLIST automatically displays a custom selection list and returns the user's selections. Only one user selection is allowed, and the selection list window closes automatically after the user makes a selection.

You can provide a default or initial selected value in the list by assigning a value to *selection* before executing SHOWLIST. If *selection* contains a value that corresponds to one of the *item* arguments when SHOWLIST executes, then that selection is designated as selected when the selection list is displayed.

If a user closes the selection list window without making a selection, SHOWLIST returns a blank value unless there was an initial value for *selection* before SHOWLIST was called.

Example

Open a selection list window that displays a list of three colors:

```
color='BLUE';
color=showlist
('RED','BLUE','GREEN','Please select a color.');
```

Because a value is assigned to the variable COLOR before SHOWLIST executes, and the value **`BLUE`** is one of the arguments for SHOWLIST, the item BLUE in the list is highlighted with an asterisk when the list is displayed.

See Also

"DATALISTC and DATALISTN" on page 326

"LISTC and LISTN" on page 518

SORT

Sorts a SAS table by one or more columns

Category: SAS Table

Syntax

sysrc=**SORT**(*table-id*,*col-list-1*<. . .*col-list-4*>);

sysrc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

table-id
contains the identifier that was assigned when the table was opened. The table must be open in UPDATE mode. If *table-id* is invalid, the program halts.
Type: Numeric

col-list
is one to four quoted strings of columns or options, separated by blanks. For a list of options, the first character in the list must be a forward slash (`/`). Columns and options cannot be mixed in the same list.
Type: Character

Details SORT uses the sorting program that SAS supports on your operating system. If a column list contains more than one column, the table is sorted by those columns in the order in which they are specified.

You can use the following options for *col-list*, depending on your operating system:

```
DATECOPY
DIAG
EQUALS
FORCE
LEAVE
LIST
MESSAGE
NODUPKEY
NODUPLICATES
NOEQUALS
OUT
REVERSE
SORTSEQ=ASCII|EBCDIC|DANISH|FINNISH
   |NATIONAL|NORWEGIAN|SWEDISH
SORTSIZE
SORTWKNO
TAGSORT
TRANTAB
```

SAS views cannot be sorted in place. To sort views, you must specify an output SAS table.

If a SAS table is already in sorted order, then you must use the FORCE option.

Note: Unless you specify the FORCE option, the SORT function and PROC SORT do not sort and replace an indexed SAS table, because sorting destroys indexes for the table. △

Example

Use the SORT function with the options NODUPKEY, NODUPLICATES, and TAGSORT to sort the SAS table MYDATA by column A in ascending order and by column B in descending order:

```
rc=sort(mydataid,'a descending b',
'/ nodupkey noduplicates tagsort');
```

SORTLIST

Sorts the items in an SCL list by value or by name

Category: List

Syntax

rc=**SORTLIST**(*list-id*<,*options*<,*start-index*<,*n-items*>>>);

rc
contains the identifier of the sorted list. The value passed as *list-id* is returned unless there is an error. The value 0 means out of memory.
Type: Numeric

list-id
is the identifier of the list to sort. An invalid *list-id* produces an error condition.
Type: Numeric or List

options
specify how the sort operation is performed. Multiple options can be specified, separated by blanks. Each option can be abbreviated to a unique substring. The substring can be as short as the first character for all options except **'NAME'** and **'NODUP'**, which may be abbreviated to two characters, **'NA'** or **'NO'**, respectively. Later keywords override previous keywords.

'ASCENDING'
Sort the list in ascending order. (This is a default.)

'DESCENDING'
Sort the list in descending order.

'NOHONORCASE'
Ignore case when comparing string values. Case is always ignored when sorting by name, because names are always converted to uppercase.

'NAME'
Sort the list by item name. Unnamed items appear before named items in an ascending sort.

'NODUP'
Delete duplicate items when sorting. All but the first item in the sort range that have the same value (or the same name, if sorting by name) are deleted. The default is not to delete duplicates.

'OBEYCASE'
Obey case when comparing string values. This is the default when sorting by value.

'VALUE'
Sort the list by item value. In an ascending sort, character items precede list identifiers, which precede numeric missing values, followed by non-missing numeric values. (This is a default.)
Type: Character

start-index
specifies the starting position for sorting a range of items in the list. By default, *start-index* is 1 (the first item). If *start-index* is positive, then the range begins *start-index* items from the beginning of the list. If *start-index* is negative, then the range begins at the item specified by ABS(*start-index*) items from the end of the list. An error condition results if the absolute value of *start-index* is zero or if it is greater than the number of items in the list.
Type: Numeric

n-items
specifies the number of items in the list to sort. The default is all items between *start-index* and the opposite end of the list. To explicitly specify all items, specify –1.
Type: Numeric

Details SORTLIST does not make a copy of the list before it is sorted. The list is modified in place.

Sublists that are contained in the sorted list are not sorted recursively.

When you specify the **'NODUP'** and **'NOHONORCASE'** options, the character list items or names that are spelled the same but differ only in case are considered duplicates, and all but the first occurrence are removed from the sorted list.

An error occurs if the list has the NOUPDATE attribute or if an item to be removed has the NODELETE attribute, if NODUP is specified. Use HASATTR to check the attributes of a list or item. To change attributes, use SETLATTR.

Examples

- □ Sort the first 10 items in a list in descending order:

```
list=sortlist(list,'D',1,10);
```

- □ Sort the last 16 items in a list in ascending order:

```
list=sortlist(list,'',-1,16);
```

- □ Sort the second ten items in a list in ascending name order, deleting items that have duplicate names:

```
list=sortlist(list,'NODUP ASCENDING NAME',11,10);
```

See Also

"HASATTR" on page 458

"REVLIST" on page 617

"ROTLIST" on page 622

"SETLATTR" on page 656

STATUS

Contains the status of program execution or overrides the normal flow of control

Category: System Variable

Details _STATUS_ is a system variable that is created for every SCL program that is compiled. The compiler creates a space for _STATUS_ in the SCL data vector. _STATUS_ is maintained as a single-character string variable.

When an SCL program executes, _STATUS_ can have one of the following values:

' '(blank)	A control or field was modified or selected.
E	An END or equivalent command was issued.
C	A CANCEL or equivalent command was issued.
P	A pop-up menu event occurred.
G	The getrow section was called for the top row of an extended table.
K	A command other than an END or CANCEL, or their equivalents, was issued. Valid only for FRAME entries.

D	A control was selected with a double click. Valid only for FRAME entries.

In addition to the execution values, you can assign the following values to _STATUS_:

'H'	Terminate the current window without further input from the user. Control returns to the program or window that invoked the application. Note that the TERM section of the program is not executed in this case. In FSEDIT, if a user modified a table variable value in the current row, the modified values are not written to the SAS table.
'R'	Resume execution of the SCL program without exiting the application in SAS/AF or the current row in FSEDIT. When you set the value of the _STATUS_ variable to 'R', the procedure ignores the END or CANCEL command that the user just issued. This value is useful only when set in the TERM section of your program or in the _preterm method of a FRAME entry control, because the specified action (not allowing an exit from the program or the current row) occurs after the user has issued an END or CANCEL command in SAS/AF or after a user has attempted to leave a row in FSEDIT.

Assigning a value to _STATUS_ does not imply an immediate return. The value of _STATUS_ is queried only after the SCL program returns control to the application. To return control to the application after assigning a value to _STATUS_, use the STOP or RETURN statement.

FRAME entries can also use the _getStatus and _setStatus methods to query and update the _STATUS_ variable.

Example

The following program calls OKTOEND.FRAME to display a confirmation window that allows the user to select 'ok' or 'cancel' in response to the END command. OKTOEND returns a 1 if it is OK to end or a 0 if it is not.

```
TERM:
 /* Check whether the END command was issued */
  if _status_ eq 'E' then
     do;
     call display( 'oktoend.frame', ok );
 /* Check whether the user wants to cancel the */
 /* END command */
      if ok eq 0 then
         do;
           _status_ = 'R';
          return;
      end;
   end;
   ...The rest of the TERM section...
return;
```

See Also

"_EVENT_" on page 379

STDMSG

Displays the text of the SAS software message that is generated by an unknown command

Category: Message

Syntax

cval=**STDMSG**();

cval
contains the message text. A blank means that no window exists in which to display the message.
Type: Character

Example

Use WORD to read a command from the command line in a PROGRAM entry and check the command for validity. If the command is not valid, the standard message is displayed. Valid commands in this case are PRINTIT and FILEIT. Any other commands produce the standard error message for invalid commands.

```
INIT:
   control always;
return;
MAIN:
   if _status_ in ('C' 'E') then return;
   command=word(1);
   call nextcmd();
   select(upcase(command));
      when('PRINTIT') _msg_='PRINTIT is specified';
      when('FILEIT') _msg_='FILEIT is specified';
      otherwise do;
         call execcmdi(command);
         stdmsg=stdmsg();
         if stdmsg ne _blank_ then _msg_=stdmsg;
         return;
         end;
   end;
return;
TERM:
return;
```

See Also

“NEXTCMD” on page 567

“NEXTWORD” on page 568

“SYSMSG” on page 686

“WORD” on page 725

STOP

Stops executing statements in the program section that is currently executing

Category: Control Flow

Comparisons: SAS Statement with limitations in SCL

Syntax

STOP;

Example

The chain of execution is begun by executing OBJ1 and includes LAB1 and LAB2. When the STOP statement is executed, the chain of execution ends, and the two PUT statements are never executed. Control is passed to MAIN.

```
INIT:
   control label;
return;
MAIN:
   put 'in MAIN';
return;

OBJ1:
   link LAB1;
   put 'after link to LAB1';
return;

LAB1:
   link LAB2;
   put 'after link to LAB2';
return;

LAB2:
   stop;
return;

TERM:
return;
```

See Also

"_STATUS_" on page 671

STRATTR

Defines a string for color and display attributes

Category: Widget or Field

Syntax

cval=**STRATTR**(*color*,*attribute*,*start*,*length*);

cval
is a string that contains the specified color and display attributes.
Type: Character

color
is a color name, or `''`to retain the current color. Colors are BLACK, BLUE, BROWN, CYAN, GRAY, GREEN, MAGENTA, ORANGE, PINK, RED, WHITE, and YELLOW. SASCOLOR window elements can also be used for *color*.
Type: Character

attribute
is a display attribute, or `''`to retain the current attribute. Attributes are NONE, BLINKING, HIGHLIGHT, HIREV, REVERSE, and UNDERLINE. If you specify a SASCOLOR window element for *color*, then *attribute* is ignored, because the SASCOLOR window element contains a display attribute. However, you must specify a placeholder (`''`) for *attribute* when you specify arguments after it.
Type: Character

start
is the starting character position to store in the attribute string.
Type: Numeric

length
is the number of character positions to store in the attribute string.
Type: Numeric

Details STRATTR defines a string that you can use with FLDATTR to change the color and display attributes of fields or portions of fields in an application window. STRATTR can be called multiple times to create a string with multiple attributes in it.

Characters whose positions are after *start* + *length* - 1 do not change color or attributes. Characters whose positions are before the start position must be initialized to the special hexadecimal character 'FF'x in order to maintain their current color and attribute. For more information about using hexadecimal characters, see "FLDATTR" on page 414.

To change the color for an entire text entry widget or field, use the FIELD function.

Example

Define an attribute string named STR that contains red reverse in the first half of the string and blue highlight in the second half. Apply the attribute string to the field ABC.

```
half=mlength(abc)/2;
str=strattr('red','reverse',1,half);
str=strattr('blue','highlight',half+1,half);
rc=fldattr('abc',str);
```

See Also

"FLDATTR" on page 414

"FLDCOLOR" on page 416

SUBMIT

Submits statements or commands to SAS for execution

Category: Submit Block

Syntax

SUBMIT<*when* <*where*>><*host*> <'STATUS'>;

when

specifies when to submit the generated statements or commands for execution and what action, if any, the procedure must take. If an option is not specified, the SUBMIT block statements are collected by SCL in a PREVIEW buffer. Options are

CONTINUE

specifies that at the end of the current submit block, the procedure submits all statements stored in the PREVIEW window and returns control to the SCL program. Execution of the program continues with the statement that follows ENDSUBMIT.

IMMEDIATE

specifies that at the end of the current submit block, the procedure submits all statements stored in the PREVIEW window and returns control to the procedure. You cannot use this option with FRAME entries, because its action could prevent the execution of other labeled sections.

Note: This means that any statements following ENDSUBMIT are not executed on this pass through the SCL program. To execute the statements following ENDSUBMIT, use conditional logic to branch around the SUBMIT IMMEDIATE statement. △

PRIMARY

specifies that at the end of the current submit block, the procedure submits all SAS statements stored in the PREVIEW window, and the user is returned to the primary window (the entry specified with the CATALOG= option in the AF command). If the current entry is the primary entry for the application, then this option restarts the current entry.

Note: This means that any statements following ENDSUBMIT are not executed on this pass through the SCL program. To execute the statements following ENDSUBMIT, use conditional logic to branch around the SUBMIT PRIMARY statement. △

TERMINATE
specifies that at the end of the current submit block, the procedure submits all SAS statements stored in the PREVIEW window and closes the AF window.

Note: The IMMEDIATE, PRIMARY, and TERMINATE options are honored only if the *where* option is not specified, that is, only if statements are to be submitted to the SAS system for execution. △
Type: Character

where
determines where statements are submitted for execution. If this option is omitted, the statements are submitted to SAS for execution. Options are

COMMAND
submits command line commands to SAS for execution. You can specify multiple commands by separating them with semicolons. You must specify the CONTINUE option along with the COMMAND option.

EDIT
submits the statements to the Program Editor window.

Note: In Version 8, using the RECALL command from the Program Editor window does not recall code that was submitted from SCL to the Program Editor. △

SQL
submits the statements to SQL for processing from both TESTAF and AF modes.

Note: The COMMAND, EDIT, and SQL options work only with the CONTINUE option. △
Type: Character

host
provides instructions for submitting the code on a particular operating system. Options are

LOCAL
executes on the current system. (This is the default.)

REMOTE
executes on a remote host. Use this option with the CONTINUE option.
Type: Character

'STATUS'
is the instruction to display the status window at all times. Use this option with the CONTINUE option.
Type: Character

Details SUBMIT labels the beginning of a block of SAS statements or commands to submit to SAS software. When SUBMIT is encountered, SCL collects all the text between SUBMIT and ENDSUBMIT and places it in the PREVIEW buffer. Based on the value of the *when* option, SCL submits the statements to SAS for execution at the appropriate time.

The *where* and *host* options are valid only if the *when* option is specified. Regardless of the value of the *where* option, all submit block text can be buffered in the PREVIEW window. You need to make certain that the CONTINUE option is specified when you want the statements to be submitted.

By default, when control returns to SCL, the program continues to execute the statements that follow ENDSUBMIT.

You can use the REMOTE command to override the *host* option of a SUBMIT statement. That is, in the SUBMIT statement, you can choose not to use the *host* option

REMOTE and can instead use the REMOTE command to control whether the generated code is executed on the local or remote host. For more information about the REMOTE command, see “Submitting Statements to a Remote Host” on page 84.

The ASIS option in the CONTROL statement allows a submit block to be submitted without formatting. Use CONTROL ASIS before the SUBMIT block to submit code that contains a DATALINES statement.

When an AF application that is invoked from an autoexec file uses both SUBMIT CONTINUE REMOTE and LOCAL blocks, the order of execution may not be correct. In this situation, you can avoid problems by using the following in place of the SUBMIT CONTINUE REMOTE blocks:

```
submit continue;
   rsubmit;
   ...SCL statements...
   endrsubmit;
endsubmit;
```

Note: If CONTROL LABEL is specified, a window variable section must not contain a SUBMIT IMMEDIATE block. △

Examples

- Submit a simple SAS program that invokes the PRINT procedure for a previously referenced SAS table:

  ```
  submit continue;
     proc print;
     run;
  endsubmit;
  ```

- Submit the LIBNAME command:

  ```
  main:
     submit command continue;
        libname;
     endsubmit;
  return;
  ```

See Also

“ENDCLASS” on page 366

“PREVIEW” on page 599

SUBMITCLEAR

Aborts a pending submit transaction

Category: Submit Block

Syntax

sysrc=**SUBMITCLEAR**();

sysrc
contains the return code for the operation:

0	successful
!0	not successful

Type: Numeric

Details SUMBITCLEAR aborts a pending submit transaction. A submit transaction may be pending if the submitted DATA steps or procedure statements are not complete, or if the submitted text contains an open quoted string or other syntax errors. SUBMITCLEAR is particularly useful for terminating programs that are waiting on input because of unmatched quotes or other syntax errors.

A submit transaction starts as SAS/AF sends submitted text in the PREVIEW buffer to SAS for execution (usually through the SUBMIT CONTINUE option). If SUBMIT requests were queued up, they would be flushed out without being executed. The currently running step is aborted, and as a result, the log will not be updated with any specific status information relating to the step that is being terminated.

Examples

Example 1: Abort a Pending Transaction Submit code to SAS for execution, then call SUBMITCLEAR to ensure there is no hanging transaction due to mismatched quoted string or other syntax errors.

```
SUBUMIT CONTINUE;
  data test;
    . . .
ENDSUBMIT;

rc = SUBMITCLEAR();
```

See Also

"PREVIEW" on page 599

"SUBMIT" on page 676

SUPAPPLY

Invokes the inherited definition of a method and passes the method's arguments in an SCL list

Category: Object Oriented

Syntax

CALL SUPAPPLY(*object-id*,*method-name*,*arg-list-id*);

return-value=**SUPAPPLY**(*object-id*,*method-name*,*arg-list-id*);

object-id
contains the identifier of the object for which the method is invoked.
Type: Numeric or Object

method-name
is the name of the method to invoke.
Type: Character

arg-list-id
is the identifier of a list of arguments that are required by the method. An invalid *arg-list-id* produces an error condition.
Type: Numeric

return-value
contains the value returned by *method-name*. The data type for *return-value* should match the data type for the called method.
Type: Numeric, Character, List, Object-name, Class, or Interface

Details SUPAPPLY provides the same functionality as SUPER except that you can pass arguments to inherited methods in an SCL list. You use SUPAPPLY to execute an inherited method when you define another method that performs additional actions. A method that calls an inherited method and includes additional actions is called an overloaded method. See "Overloading Methods" on page 108 for more information.

In Version 8, you must use dot notation to call overloaded methods. Dot notation provides compiler time checking and better performance at run time. With overloaded methods, the compiler must be able to check method signatures in order to call the correct method. For details about dot notation, see "Accessing Object Attributes and Methods With Dot Notation" on page 119.

You can use SUPAPPLY as a function if the called method returns a value with a RETURN statement in the program that defines the method.

Although the method name is typically the name of the currently executing method, which is stored in the system variable _METHOD_, any other method name can be used.

The object identified by *object-id* must be the same object whose method is currently executing. The identifier for this object is stored in the system variable _SELF_. In methods defined in a CLASS or USECLASS statement block, all references to the class methods and attributes can bypass references to _SELF_.*attribute* and _SELF_.*method*(...). For example, to call a super method with dot notation in a method definition, you can use **supapply();** , which is equivalent to **call supapply(_self_,'m1');**.

Example

Consider an Object class in which a Transaction method receives a variable-length parameter list of transactions to record in a table. A subclass of this class records the transactions in an audit trail table that contains two numeric variables, DATETIME and TCOUNT. These variables record the date/time and the number of transactions processed. This example shows the Transaction method, which invokes the inherited Transaction method and then records the size of the transaction (the number of items in the argument list) for a table. The object has the attributes **audit**, **tc_vmum**, and **dt_vnum**. **Audit** is a table ID for an audit table. **Tc_vnum** is the variable number for the TCOUNT variable. **Dt_vnum** is the variable number for the DATETIME variable.

```
useclass lib.cat.myclass.class;

/* TRANSACT.SCL: TRANSACTION method */
Transaction: method arglist= transactions;
   call supapply(_self_,'Transaction',transactions);
   if audit then do;
      nTransactions=listlen(transactions);
      call putvarn(audit,tc_vnum,nTransactions);
```

```
        call putvarn(audit,dt_vnum,datetime());
        rc=update(audit);
     end;
  endmethod;
  enduseclass;
```

This method can be invoked with an arbitrary number of transactions using dot notation, where the SCL variable **`listOftransactions`** is an SCL list that contains one or more transaction objects.

```
  dcl lib.cat.myclass.class obj=_NEW_lib.cat.myclass();

  obj.Transaction(t1,t2,t3,t4,t5);
  obj.Transaction(t1,t2);
  obj.Transaction(listOftransactions);
```

See Also

"APPLY" on page 252

"INSTANCE" on page 485

"METHOD" on page 540

"NOTIFY" on page 572

"SEND" on page 644

"SUPER" on page 681

SUPER

Invokes the inherited definition of a method

Category: Object Oriented

Syntax

CALL SUPER(*object-id*,*method-name*<,*parameters*>);

<*return-value*=>**_SUPER**<.*method-name*>(<*parameters*>);

object-id
contains the identifier of the object for which the method is invoked.
Type: Numeric or Object

method-name
is the name of the method to invoke.
Type: Character

parameters
are additional numeric or character arguments that are required by the method. Use commas to separate multiple options.

Note: These parameters are update parameters. See "Input, Output, and Update Parameters" on page 34 for more information. △
Type: Numeric, Character

return-value
contains the value returned by the inherited method.
Type: Numeric, Character, List, Object-name, Class, or Interface

Details SUPER provides a convenient way of calling a method in the parent class from a method in the child class. In particular, it is useful for calling the corresponding parent method from an overridden method in the child.

Although *method-name* is typically the name of the currently executing method, which is stored in the system variable _METHOD_, any other method name can be used.

The object identified by *object-id* must be the same object whose method is currently executing. The identifier for this object is stored in the system variable _SELF_.

For more information about system variables, see "System Variables" on page 24.

SUPER can also be used with CLASS blocks, USECLASS blocks and dot notation by using _SUPER. If you are specifying the same method in the parent and child classes, you do not need to specify the method name.

Examples

Example 1: Calling the Parent Method of an Overridden Method In this example, _SUPER is used to call the parent method of an overridden method in class X.

```
Y.SCL
  class y;
    m: method;
      ...SCL statements...
    endmethod;
  endclass;

X.SCL
  class x extends y;
    m: method;
      _super();
      /* _super invokes method M in class Y */
    endmethod;
  endclass;
```

Example 2: Calling a Different Method in the Parent Class To call a different method in the parent class, specify the method name after _SUPER using dot notation. The following example invokes the M2 method in class Y using _SUPER:

```
Y.SCL
  class y;
    m2: method;
      ...SCL statements...
    endmethod;
  endclass;

X.SCL
  class x extends y;
    m: method;
      _super.m2();
    endmethod;
  endclass;
```

Example 3: Calling a Method in the Parent of a Parent Class This example demonstrates how you can use inheritance to invoke a method in the parent of a parent class.

```
S.SCL
  class s;
    m: method n: num return=num;
      dcl num x;
      x=n+199;
      return x;
    endmethod;
  endclass;

S2.SCL
  class s2 extends s;
    m: method n: num return=num/(state='O');
      dcl num x;
      x=n+_super(1);
      return x;
    endmethod;
  endclass;

S3.SCL
  class s3 extends s2;
    n: method return=num;
      dcl num x;
      x=_super.m(-10);
      return x;
    endmethod;
  endclass;

DRS.SCL
init:
  dcl s3 sobj=_new_ s3();
  dcl num x;
  dcl string s;
  x=sobj.n();
  put x=;
return;
```

This example results in the following output:

```
x=190
```

The calling sequence for the above example is as follows:

1. Method N in class S3 is invoked in DRS.SCL.
2. In N, method M is invoked in class S2 via _SUPER. The parameter is **-10**.
3. Method M, which is within S2, invokes method M in class S via _SUPER. The parameter is **1**.
4. In S, **1** is added to **199** and returned to S2.
5. In S2, **200** is added to **-10** and returned to S3.
6. In S3, **190** is returned to DRS.SCL.

See Also

"APPLY" on page 252

"INSTANCE" on page 485

"METHOD" on page 540

"NOTIFY" on page 572

"SEND" on page 644

"SUPAPPLY" on page 679

SYMGET and SYMGETN

Return the value stored in a macro variable

Category: Macro

Syntax

cval=**SYMGET**(*macro-var*);

nval=**SYMGETN**(*macro-var*);

cval
contains the character value returned by SYMGET.
Type: Character

nval
contains the numeric value returned by SYMGETN.
Type: Numeric

macro-var
is the macro variable.
Type: Character

Details SYMGET and SYMGETN return values of macro variables when the program executes, whereas "&*macro-var*" returns values when the program compiles. SYMGET returns the value of a SAS macro variable as a character value and SYMGETN returns the value as a numeric value.

Examples

Example 1: Using the SYMGET Function Execute commands if the operating system is Windows:

```
if symget('SYSSCP') = 'WIN' then do;
   rc = optsetn('xwait',0);
   rc = optsetn('xsync',0);
end;
```

Example 2: Using the SYMGETN Function Return the value of the macro variable UNIT at program execution time:

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

Example 3: Returning Macro Values at Program Compile Time Return the values of the macro variables UNIT and SYSJOBID at program compile time:

```
nval=&unit;
cval="&sysjobid";
```

See Also

"SYMPUT and SYMPUTN" on page 685

SYMPUT and SYMPUTN

Store a value in a SAS macro variable

Category: Macro

Syntax

CALL SYMPUT(*macro-var*,*cval*);

CALL SYMPUTN(*macro-var*,*nval*);

macro-var
is the macro variable to store a value in.
Type: Character

cval
is the character value for SYMPUT to store in *macro-var*.
Type: Character

nval
is the numeric value for SYMPUTN to store in *macro-var*.
Type: Numeric

Details SYMPUT stores a character value in a SAS macro variable, whereas SYMPUTN stores a numeric value in a SAS macro variable. If *macro-variable* does not exist, SYMPUT and SYMPUTN create it. SYMPUT and SYMPUTN make a macro variable assignment when the program executes.

Examples

Example 1: Using the SYMPUT Routine Store the value of the SCL variable SCLVAR in the macro variable TBL:

```
call symput('tbl',sclvar);
```

Example 2: Using the SYMPUTN Routine Store the numeric value 1000 in the macro variable UNIT:

```
call symputn('unit',1000);
```

See Also

"SYMGET and SYMGETN" on page 684

SYSMSG

Returns the text of SCL error messages or warning messages

Category: Message

Syntax

cval=**SYSMSG**();

cval
contains the text of the SCL error message.
Type: Character

Details SYSMSG returns the text of error messages or warning messages that are produced when SCL 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. Therefore, if you subsequently call SYSMSG before another error condition occurs, it returns a blank value.

Example

Display the system error message that is generated if FETCH cannot copy the next row into the Table Data Vector for the SAS table identified by the value stored in DSID. The return code is 0 only when a next record is successfully fetched.

```
rc=fetch(dsid);
if rc ne 0 then _msg_=sysmsg();
```

See Also

"SYSRC" on page 686

"_MSG_" on page 554

SYSRC

Returns a system error number or the exit status of the most recently called entry

Category: Message

Syntax

rc=**SYSRC**(<*display-stat*>);

rc
contains the return code for the most recent error or warning condition, if *display-stat* is omitted. Otherwise, returns the exit status of CALL DISPLAY:

0	A user used an END command.
–1	A user used a CANCEL command.

Type: Numeric

display-stat
causes the SYSRC function to return the most recent exit status of CALL DISPLAY. The value of the argument does not matter, only whether any value is specified.
Type: Numeric

Details If you pass an argument to SYSRC, the function returns the exit status of the most recently called execution of CALL DISPLAY rather than the SAS system return code for the most recent error or warning condition. Thus, you can use SYSRC to determine how a user terminated an entry that was called with CALL DISPLAY.

To return the SAS system return code, do not specify any value for *display-stat*. See Chapter 14, “SAS System Return Codes,” on page 227for more information about how to use return code values.

Example

Determine how a user exited from another entry that was called within the current application:

```
call display('test.scl');
if sysrc(1)=-1 then
 _msg_=
    'User exited TEST.SCL with a CANCEL command';
else
 _msg_=
    'User exited TEST.SCL with an END command';
```

See Also

“SYSMSG” on page 686

SYSTEM

Issues a host system command

Category: Command

Syntax

rc=**SYSTEM**(*cval*);

rc
contains the host system return code for the operation.
Type: Numeric

cval
is the host command to be executed. To enter the host command processor for your system, specify a blank string (' ').
Type: Character

Details Using SYSTEM is equivalent to using the X command for issuing a system command. The action that takes place depends on which command you specify. The window may be temporarily overwritten due to the actions of the command. The commands that can be issued are operating-system dependent.

Example

Issue the DIR command to the host operating system:

```
rc=system('dir');
if (rc) then _msg_=
   'Failed to execute the DIR command.';
```

THROW

Raises an exception

Category: Control Flow

Syntax

THROW *exception*;

exception
is the local variable for the exception that you want to process.

Details

All exceptions are subclasses of the SCLException class. You can use the CLASS statement to define your own exception classes, and then use THROW statements to raise the exceptions.

When an exception is thrown, normal execution of the SCL entry stops, and SCL begins looking for a CATCH block to process the exception.

Note: You must always declare a variable to hold the thrown exception. △

Example

The following class defines a subclass of SCLException called NewException, which defines an attribute named SecondaryMessage:

```
Class NewException extends SCLException
   dcl string SecondaryMessage;
endclass;
```

You can create a new instance of NewException and raise this exception with the THROW statement, as shown here in the ThrowIt class:

```
Class ThrowIt;
  m: method;
  dcl NewException NE = _new_ NewException('Exception in method m');
  NE.SecondaryMessage = "There's no code in m!";
  throw NE;
  endmethod;
endclass;
```

See Also

"CATCH" on page 268

Chapter 10, "Handling Exceptions," on page 157

TOPROW

Scrolls a row to the top of an extended table

Category: Extended Table

Syntax

CALL TOPROW(*row*);

row
is the number of the table row to be scrolled to the top of the table.
Type: Numeric

Details The TOPROW routine cannot be called in the getrow or putrow section of an SCL program.

You can use TOPROW only on extended tables in PROGRAM entries. Because you can define extended tables only in SAS/AF software, you cannot use TOPROW in FSEDIT or FSVIEW programs.

Examples

Example 1: Specifying Which Row to Scroll

Scroll the fifth row to the top of the table:

```
call toprow(5);
```

Example 2: Letting the Program Determine Which Row to Scroll

Suppose you have a PROGRAM entry window that contains three character fields:

1 VALUE, in the non-scrollable area of the window. Turn the CAPS attribute off for VALUE.
2 NAME, the first field in the extended table's logical row. Turn the CAPS attribute off for NAME.

3 GENDER, the second field in the extended table's logical row.

When a user enters a name in **VALUE**, the table scrolls so that the corresponding row is at the top of the table.

This program controls the window:

```
INIT:
  dsid=open('sasuser.class');
  call set(dsid);
  call setrow(0,0,'','y');
  vnum=varnum(dsid,'name');
return;
MAIN:
rc=
where(dsid,"name contains '"||value|| "'");
 any=attrn(dsid,'any');
  if any then do;
    rc=fetch(dsid);
    firstmatch=getvarc(dsid,vnum);
    rc=where(dsid);
    recnum=locatec(dsid,vnum, firstmatch);
    call toprow(recnum);
  end;
return;

TERM:
  if dsid then dsid=close(dsid);
return;

getrow:
  if fetchobs(dsid,_currow_)=-1
   then call endtable();
return;
```

See Also

"CURTOP" on page 324

"ENDTABLE" on page 368

"SETROW" on page 664

TRACEBACK

Displays traceback information for an SCL execution stack

Category: Utility

Syntax

CALL TRACEBACK(<*list-id*><,'_ALL_'>;

list-id
contains the identifier of the SCL list to store traceback information in. You must create the list before passing it to TRACEBACK. An invalid *list-id* produces an error condition.
Type: Numeric or List

ALL
is the instruction to display the link stack in addition to the SCL execution stack.
Type: Character

Details The execution stack consists of the program that is currently being executed, plus all programs that were called to display the current program. TRACEBACK displays the execution stack as a list of entry names and associated line numbers. The _ALL_ argument displays the *link stack*, which lists the labeled sections that are called within the program. The link stack does not include labeled sections that are called with a GOTO statement. The line number for an entry indicates where the entry transferred control to the next entry in the list.

Examples

The following examples use three SCL entries, START.SCL, ANALYZE.SCL, and REPORT.SCL. START.SCL contains a link at line 5 and calls ANALYZE.SCL at line 9. ANALYZE.SCL contains links at line 5, 11, and 16 and calls REPORT.FRAME at line 21.

Example 1: Tracing Program Execution The TRACEBACK routine is executed at line 6 of REPORT.SCL.

```
call traceback();
```

The following traceback list is printed in the LOG window:

```
In routine: LIB.TEST.REPORT.SCL line 6
Called from LIB.TEST.ANALYZE.SCL line 21
Arguments passed to DISPLAY:
  1 (Character Literal) = 'report.scl'
Called from LIB.TEST.START.SCL line 9
Arguments passed to DISPLAY:
  1 (Character Literal) = 'analyze.scl'
```

Example 2: Saving Traceback Information in an SCL List Change REPORT.SCL to save traceback information in an SCL list, and then display the traceback list:

```
tb = makelist();
call traceback(tb);    /* line 7 */
call putlist(tb, 'Traceback:', 0);
tb = dellist(tb);
```

This program produces the following output:

```
Traceback:(LIB.TEST.REPORT.SCL=7
           LIB.TEST.ANALYZE.SCL=21
           LIB.TEST.START.SCL=9
           )[1905]
```

Example 3: Tracing the Execution Stack and Program Links Change REPORT.SCL to execute a traceback, using the _ALL_ argument on line 6:

```
call traceback(0,'_all_');
```

The following traceback list is printed in the LOG window:

```
In routine: LIB.TEST.REPORT.SCL line 6
Called from LIB.TEST.ANALYZE.SCL line 21
Arguments passed to DISPLAY:
 (Character Literal) = 'report.scl'
Linked from LIB.TEST.ANALYZE.SCL line 16
Linked from LIB.TEST.ANALYZE.SCL line 11
Linked from LIB.TEST.ANALYZE.SCL line 5
Called from LIB.TEST.START.SCL line 9
Arguments passed to DISPLAY:
  1 (Character Literal) = 'analyze.scl'
 Linked from LIB.TEST.START.SCL line 5
```

Example 4: Comparing Tracebacks With and Without _ALL_ Change REPORT.SCL to execute the traceback without the _ALL_ option at line 8 and to execute the traceback with the _ALL_ option at line 9:

```
listid1=makelist();
listid2=makelist();
call traceback(listid1);
call traceback(listid2,'_ALL_');
call putlist(listid1,
   'Traceback without LINK stack=',0);
call putlist(listid2,
   'Traceback with LINK stack=',0);
listid1=dellist(listid1);
listid2=dellist(listid2);
```

This program produces the following output:

```
Traceback without LINK stack=(LIB.TEST.REPORT.SCL=8
LIB.TEST.ANALYZE.SCL=21
LIB.TEST.START.SCL=9
                             )[1905]
Traceback with LINK stack=(
 LIB.TEST.REPORT.SCL=9
 LIB.TEST.ANALYZE.SCL=21
 LIB.TEST.ANALYZE.SCL=16
 LIB.TEST.ANALYZE.SCL=11
 LIB.TEST.ANALYZE.SCL=5
 LIB.TEST.START.SCL=9
 LIB.TEST.START.SCL=5
                          )[1907]
```

Note: 541, 1905, and 1907 are the list identifiers that were assigned when these examples were run and may be different each time the examples are run. △

See Also

"MAKELIST" on page 535

"TRACEBACK" on page 223

UNGRAY

Ungrays a window element

Category: Control or Field

Syntax

rc=**UNGRAY**(*var-name*<,*station*<,*row*>>);

rc

contains the return code for the operation:

0	successful
≠0	unsuccessful

Type: Numeric

var-name

is the control or choice group to be ungrayed.

Type: Character

station

is the number of a field within the choice group or the number of a button within the radio box.

Type: Numeric

row

is the number of a row when the choice group is in the scrollable section of an extended table. The *row* parameter is valid for PROGRAM entries but not for FRAME entries. Specify *row* only when you want to ungray a station from outside the extended table's getrow or putrow section.

Type: Numeric

Details You can use UNGRAY along with GRAY to control the availability of a choice group, a station, or a window element, based on the program flow.

When a window element is ungrayed, it becomes unprotected and reverts to its normal color. An ungrayed window element can once again receive input such as mouse clicks. FRAME entry controls can also use the _ungray method.

Example

Make a station available only when the value of variable DEPT is ADMIN:

```
if (dept='ADMIN') then
      rc=ungray('personal',3);
   else
      rc=gray('personal',3);
```

See Also

"ACTIVATE" on page 249

"GRAY" on page 457

"ISACTIVE" on page 492

"ISGRAY" on page 493

UNIQUENUM

Returns a unique number

Category: Utility

Syntax

num=**UNIQUENUM**();

num
contains the unique number that is returned by the function.
Type: Numeric

Details UNIQUENUM returns a number that is unique for each call to the function during a SAS session.

Example

```
num=uniquenum();
put num=;
```

UNLOCK

Releases a lock on the current row

Category: SAS Table

Syntax

sysrc=**UNLOCK**(*table-id*);

sysrc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

table-id
is the identifier that was assigned when the table was opened. If *table-id* is invalid, the program halts.
Type: Numeric

Details A table that is opened in UPDATE mode receives RECORD-level locking by default. Whenever an application reads a row from a table that was opened in UPDATE mode, it attempts to obtain a lock on the row. All of the following functions lock a row when the table is opened in UPDATE mode:

DATALISTC

DATALISTN

FETCH

FETCHOBS

LOCATEC

LOCATEN

Row locks are implicitly released when a different row is read. However, when a user is finished with a row but has not read another row, you can use UNLOCK to explicitly release a lock on the row.

This function is useful when rows from a secondary SAS table are read to obtain values.

Note: UNLOCK is not directly related to the LOCK function, which locks SAS catalogs, catalog members, and SAS tables. However, if the table in question is accessed through SAS/SHARE software using the REMOTE engine, then UNLOCK can be used to enable other applications to access individual rows. See *SAS Language Reference: Dictionary* for more information. △

Example

Call FETCH to read a new row from the SAS table MYDATA, which is opened in UPDATE mode. After data from the row is processed, call UNLOCK to release the lock on the row.

```
dsid=open('mydata','u');
rc=fetch(dsid);
    ...more SCL statements...
rc=unlock(dsid);
```

See Also

"CLOSE" on page 292

"DATALISTC and DATALISTN" on page 326

"FETCH" on page 389

"FETCHOBS" on page 390

"LOCATEC and LOCATEN" on page 527

"OPEN" on page 576

UNPROTECT

Removes protection from a FRAME entry control or a field

Category: Control or Field

Syntax

UNPROTECT *wvar-names* | _ALL_;

wvar-names
lists one or more window variables to unprotect, or **_ALL_** to unprotect all window variables. To specify multiple names, use blanks to separate the names.
Type: Character

Details Use UNPROTECT to temporarily override the PROTECT attribute. The variable to be unprotected cannot be an element of an array. To unprotect an array element, use FIELD.

When used with PROTECT, UNPROTECT enables you to unprotect and protect controls and fields selectively. Thus, you can force the user to enter values in a predetermined order.

If you unprotect a window variable with the UNPROTECT statement in FSEDIT applications and then issue the MODIFY command to edit the custom screen, the PROTECT attribute is removed for this variable in the PROTECT window.

FRAME entry controls can also use the _unprotect method. Unprotecting some FRAME entry controls (block, check box, icon, list box, pushbutton, radio box, scroll bar, and slider) is the same as calling the UNGRAY function or the _ungray method.

Example

Enable users to enter values for fields that were previously protected:

```
if (modified(tablename)) then
   do;
      protect tablename;
      unprotect vars;
      cursor vars;
      _msg_=
'Enter the names of the variables to be printed.';
   end;
```

See Also

"DISPLAYED" on page 353

"ERROR" on page 375

"ERROROFF" on page 376

"ERRORON" on page 377

"FIELD" on page 394

"MODIFIED" on page 550

"PROTECT" on page 602

UNSELECT

Deselects a specified row of a selection list

Category: Extended Table

Syntax

rc=**UNSELECT**(*row*);

rc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

row
is the row number to deselect. If an invalid row number is specified, no action is taken.

Type: Numeric

Details UNSELECT is useful for forcing the deselection of a row. Normally a user selects and deselects a row by pressing ENTER or by clicking on the row with the mouse.

You can use UNSELECT only for selection lists that were built with extended tables in PROGRAM entries. Window controls must use the _unselectRow method. Because you can define extended tables only in SAS/AF software, you cannot use UNSELECT in FSEDIT or FSVIEW programs.

In order for an extended table to be considered a selection list, you must specify the number of selections in the SETROW routine.

Example

Force row 5 to be deselected:

```
rc=unselect(5);
```

See Also

"ISSEL" on page 495

"NSELECT" on page 573

"SELECT" on page 639

"SELECTED" on page 642

UPDATE

Writes values from the Table Data Vector (TDV) to the current row in a SAS table

Category: SAS Table

Syntax

sysrc=**UPDATE**(*table-id*);

sysrc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

table-id
is the identifier that was assigned when the table was opened. If *table-id* is invalid, the program halts.
Type: Numeric

Details The table must be opened in UPDATE mode. The row to be updated is the current row. To place values in the TDV, use PUTVARC, PUTVARN, or SET.

Example

Update the current row in the open SAS table whose table identifier value is stored in the column MYDATAID. If the return code, RC, is nonzero, the system error message is displayed on the message line.

```
rc=update(mydataid);
if rc then _msg_=sysmsg();
```

See Also

"APPEND" on page 251

"FETCH" on page 389

"FETCHOBS" on page 390

"GETVARC and GETVARN" on page 450

"PUTVARC and PUTVARN" on page 609

"SET" on page 646

USECLASS

Implements methods for a class and binds them to the class definition

Category: Object Oriented

Syntax

USECLASS *class-name*;
<(*method-implementation-blocks*)>

ENDUSECLASS;

class-name
is the name of a class entry that has been defined either with an SCL CLASS statement or with the AF Class Editor.

method-implementation-blocks
are SCL method implementation blocks. For detailed information, see "METHOD" on page 540 or "CLASS" on page 277.

Details The USECLASS statement binds methods that are implemented within it to a class definition. USECLASS blocks are especially useful for defining method implementations in a separate SCL entry from the one that contains the CLASS statement that defines the class. This feature enables group development of classes, because multiple people can work on class methods implementations simultaneously.

USECLASS labels the beginning of a program block that implements methods for *class-name*. The USECLASS statement block ends with the ENDUSECLASS statement. *Class-name* must be already defined in an existing CLASS entry.

The only SCL statements allowed in USECLASS blocks are METHOD implementation blocks. The signature (parameter types) of methods must match the associated method definitions in the CLASS entry. If a return type is defined, it must also match the associated method definition in the CLASS entry. If the parameter storage (INPUT/OUTPUT/UPDATE) is omitted, the SCL compiler will assume that it has the same storage specification as the associated method definition in the CLASS entry.

Method implementations inside a USECLASS block can include any SCL statements (except CLASS, USECLASS, or INTERFACE statements), or SCL functions and routines, including DECLARE to declare local variables. If you want to define a local variable that can be used across the class, you must define it as a private attribute either inside the CLASS statement or in the Class Editor. METHOD blocks can include labeled sections. However, labeled sections that are outside a method block must be re-coded as PRIVATE methods, and the LINK statements that call them must be changed to method calls. For more information, see "METHOD" on page 540.

If a local variable that is defined in a METHOD block has the same name as a class attribute, SCL gives precedence to the local variable. If a class method has the same name as any SCL-supported function, SCL gives precedence to the function. If an attribute array has the same name as a class method, SCL gives precedence to the method.

Do not declare the _SELF_, _FRAME_, _CFRAME_, _METHOD_, or _EVENT_ system variables inside a CLASS or USECLASS block. SCL automatically sets these values when it is running methods that are defined in CLASS or USECLASS blocks. Redefining any of these system variables can introduce unexpected behavior.

Because USECLASS binds the methods that it contains to the class that is defined in a CLASS entry, all references to the methods and the attributes of the class can bypass references to _SELF_.*attribute* and _SELF_.*method*(...). Because the binding occurs at compile time, the SCL compiler can detect whether an undefined variable is a local variable or a class attribute.

You can also use the _super method in method code that is inside a USECLASS statement block without having to specify either an object identifier or the method whose super method you are calling. You can use the _super method to call any method. For example, to invoke the super ADD method, you would use

```
_super.add();
```

To override the _init method, you must first call the super _init method. The _init method can be short-cut. For example:

```
_super();
```

The USECLASS statement also enables you to define implementations for overloading methods, which means that multiple methods have the same name. Methods that have the same name are allowed in a USECLASS block only if the signatures, or parameter numbers or types, are different. For example, a class can have a COMBINE method that has numeric parameters and that adds parameter values. It can also contain another COMBINE method that has character parameters and that concatenates parameter values.

Examples

Example 1: Group Development Using a USECLASS Block USECLASS can be used for group development. Suppose a group with three members ONE, TWO and THREE was assigned to develop WORK.A.GROUP.CLASS. This class is defined as follows:

```
Class work.a.group.class;
   Public  Num  nAttr;
   Public  Char cAttr;
   m1: Method n:Num / (scl='work.a.one.scl');
   m1: Method c:Char / (scl='work.a.one.scl');
   m2: Method n:num / (scl='work.a.two.scl');
   m3: Method Return=Num / (scl='work.a.two.scl');
   m4: Method Return=Char / (scl='work.a.three.scl');
EndClass;
```

Issue the SAVECLASS command to create this class. After the class is created, programmer ONE starts to create method implementations in WORK.A.ONE.SCL as follows:

```
UseClass work.a.group.class;
   m1: Method m:Num;
          nAttr = m;
       EndMethod;
   m1: Method c:Char;
          cAttr = c;
       EndMethod;
EndUseClass; +
```

Programmer ONE is also responsible for compiling and testing this method implementation before the final system integrations.

Programmer TWO creates method implementations in WORK.A.TWO.SCL as follows:

```
UseClass work.a.group.class;
    m2: Method n:Num;
          put n=;
        EndMethod;
    m3: Method Return=Num;
           return(nAttr);
        EndMethod;
EndUseClass;
```

Programmer TWO is also responsible for compiling and testing this method implementation before the final system integrations.

Programmer THREE creates method implementations in WORK.A.THREE.SCL as follows:

```
UseClass work.a.group.class;
   m4: Method return=Char;
```

```
          Return(cAttr);
       EndMethod;
EndUseClass;
```

Programmer THREE is also responsible for compiling and testing this method implementation before the final system integrations. From this example, we found that a USECLASS block can be used to simplify group development in an object-oriented development environment.

Example 2: Valid SCL Statements in a USECLASS Block This example shows that a method implementation block is the only SCL statement allowed in a USECLASS block. Other SCL statements such as the DECLARE statement are not allowed in a USECLASS block. However, these SCL statements can be written inside the method implementation block.

```
USECLASS work.classes.myclass.class;
      /* Method M1 without a signature */
   M1:  method;
         ...more SCL statements...
   endmethod;
      /* The next statements are invalid because  */
      /* they are not in a METHOD statement. They */
      /* will produce a compiler warning message. */
   DCL num n = 1; /* Invalid statement */
   n=10;   /* Invalid statement */
      /* Method M1 has a numeric parameter. */
   M1:  method n: num;
        /* Any SCL statement can be applied in the */
        /* method implementation block. */
        DCL num n;
        n = n + 1;
        ...more SCL statements...
   endmethod;
      /* Method M1 has a character parameter. */
   M1:  method s: char;
        DCL Char arr(3);
        DCL num i;
        DO i = 1 to dim(arr);
           arr(i) = s;
        END;
        ...more SCL statements...
   endmethod;
      /* The next statement is invalid because */
      /* it is not in a METHOD statement.      */
   if s=1 then
      put 'pass';
      /* Other methods */
   M2:  method return=num;
        ...more SCL statements...
        return(1);
   endmethod;
 ENDUSECLASS;
```

Example 3: Bypassing _SELF_ References in a USECLASS Block Assume that there are four class attributes defined for a class: N1 and N2 are numeric attributes, and C1 and C2 are character string attributes. Both M1 methods are class methods. One of the M1 methods takes no parameter. The other M1 method takes one numeric parameter. The private method M2 is used as a debugging routine. If the variables or methods are class attributes or class methods, short-cut syntax (see also "Referencing Class Methods or Attributes" on page 99) can be used to bypass dot notation.

```
Import Sashelp.Fsp.Colleciton.Class;
Useclass work.classes.another.class;

   _init: method / (State='O');
         /* Equivalent of _super._init();   */
         /* or call super(_self_,'_init'); */
      _super();
         /* Equivalent to _self_.n1=1  */
      n1=1;
         /* Local variable n2 dominates */
         /* class attribute n2          */
      n2=1;
         /* Use SAS function ABS  */
      n3(1)=abs(n2);
         /* Equivalent to _self_.c1='a' */
      c1='a';
         /* Local variable c2 dominates */
         /* class attribute c2          */
      c2='a';
   endmethod;

   m1: method;
         /* Equivalent to if _self_.n1>0 then */
      if n1>0 then
           /* Equivalent to _self_.n1 + 1     */
         n1+1;
      else
            /* Equivalent to          */
            /* _self_.m1(_self_.n1);  */
         m1(n1);
   endmethod;

     /* Method m1 with numeric parameter */
  m1: method n: num;
       if (n < 0) then
          n = ---n;
       /* - Invoke M1 method --- */
       m1(n);
       m2(n+n);
       DCL Collection col = _NEW_ Collection();
       /* -- Must use the dot notation here - */
       col.add(3);
   endMethod;

   /* - Private Debugging method --- */
   m2: Private Method n:Num;
         put 'Debugging ' n=;
      endMethod;

endUseClass;
```

Example 4: Subclassing TextEntry_c for FRAME Programming

The class Mywidget is defined using SCL class syntax. First, edit WORK.A.MYWIDGET.SCL and include the following source:

```
Import Sashelp.Classes;
Class work.a.Mywidget.class Extends textEntry_c;
    m1: Public Method c: char / (scl='work.a.myClass.scl');
EndClass;
```

Issue the SAVECLASS command to compile this program and create the CLASS entry WORK.A.MYWIDGET.CLASS.

Edit WORK.A.MYCLASS.SCL, using USECLASS to include the method implementations.

```
Useclass work.a.Mywidget.class;
    m1: Method c:char;
        text = c; /* Equivalent to _SELF_.text = c; */
                  /* Text is an class attribute    */
        EndMethod;
EndUseClass;
```

Compile this program.

Now edit WORK.A.B.FRAME, and in the component window add the Mywidget class to the list (via the AddClasses button on the pop-up menu). Drag the Mywidget class to the frame and in the frame's SCL source, enter

```
Init:
   Mywidget1.m1('Hello! ');
   Return;
Main;
   Return;
```

Compile and TESTAF this program. You should see Mywidget with "Hello!" in the text field. This example could also be done using only the CLASS block (without USECLASS). Just put the method implementation directly in the CLASS block.

```
Import Sashelp.Classes;
Class Mywidget Extends TextEntry_C;
    m1: Method c:Char;
         Text = c;
        EndMethod;
EndClass;
```

See Also

"ARRAY" on page 254

"CLASS" on page 277

"CREATESCL" on page 316

"DECLARE" on page 330

"ENDCLASS" on page 366

"METHOD" on page 540

VALUE

Contains the value of a FRAME entry component

Category: System Variable

Details _VALUE_ is a system variable. It is provided automatically by the FRAME entry in SAS/AF, but the SCL compiler does not automatically create a space for it in the SCL data vector. As a result, you get a warning when you compile a FRAME or SCL entry that uses _VALUE_. The warning is generated because _VALUE_ is being referenced at compile time, but no value is assigned to it until run time. You can safely ignore this warning. If you prefer to prevent the warning message from being generated, use the following assignment statement at the top of your program:

```
_value_=_value_;
```

If the FRAME entry component has a character value, then you must declare _VALUE_ as a character variable using the DECLARE or LENGTH statement.

Numeric FRAME entry components and character FRAME entry components cannot share the same methods if the methods use _VALUE_, because _VALUE_ cannot be declared as both a numeric and character variable in the same SCL entry. Doing so results in one of the following execution errors:

```
ERROR: Expecting string ($),
       received SCL number (symbol '_VALUE_').
ERROR: Expecting number (#),
       received SCL string (symbol '_VALUE_').
```

VALUE has a valid value only when a FRAME entry or a method for a FRAME entry component is running.

Example

Suppose a FRAME entry contains two Version 6 text entry controls, OBJ1 and OBJ2. OBJ1 is subclassed, and its _select method is defined as follows:

```
length _value_ text $20;
SELECT: method;
   _frame_._getWidgets('obj2',obj2);
   obj2._getText(text);
  _value_ = text;
endmethod;
```

When OBJ1 is modified, the _select method queries the object identifier for OBJ2 and retrieves its value. It then assigns that value to OBJ1 by assigning TEXT to _VALUE_.

See Also

"_SELF_" on page 643

"_METHOD_" on page 548

VARFMT

Returns the format that is assigned to a SAS table column

Category: Variable

Syntax

format=**VARFMT**(*table-id*,*var-num*);

format
contains the format that was assigned to the specified column.
Type: Character

table-id
is the identifier that was assigned when the table was opened. If *table-id* is invalid, the program halts.
Type: Numeric

var-num
is the number of the column's position in the SAS table. This number is adjacent to the column in the list that is produced by the CONTENTS procedure. The VARNUM function returns this number.
Type: Numeric

Details If no format has been assigned to the column, a blank string is returned.

Example

Obtain the format of the column NAME in the SAS table MYDATA:

```
length fmt $ 12;
dsid=open('mydata','i');
if dsid then
   do;
      fmt=varfmt(dsid,varnum(dsid,'name'));
      rc=close(dsid);
   end;
```

See Also

"VARINFMT" on page 706

"VARNUM" on page 713

VARINFMT

Returns the informat that is assigned to a SAS table column

Category: Variable

Syntax

informat=**VARINFMT**(*table-id*,*var-num*);

informat
contains the informat that is assigned to the column.
Type: Character

table-id
is the identifier that was assigned when the table was opened. If *table-id* is invalid, the program halts.
Type: Numeric

var-num
is the number of the column in the SAS table. This number is adjacent to the column in the list that is produced by the CONTENTS procedure. The VARNUM function returns this number.
Type: Numeric

Details If no informat has been assigned to the column, a blank string is returned.

Example

Obtain the informat of the column NAME in the SAS table MYDATA:

```
length infmt $ 12;
tableid=open('mydata','i');
if tableid then
   do;
      infmt=varinfmt(tableid,varnum(tableid,'name'));
      rc=close(tableid);
   end;
```

See Also

"VARFMT" on page 705

"VARNUM" on page 713

VARLABEL

Returns the label that is assigned to a SAS table column

Category: Variable

Syntax

cval=**VARLABEL**(*table-id*,*var-num*);

cval
contains the label that is assigned to the specified column.
Type: Character

table-id
is the identifier that was assigned when the table was opened. If *table-id* is invalid, the program halts.
Type: Numeric

var-num
is the number of the column in the SAS table. This number is adjacent to the column in the list that is produced by the CONTENTS procedure. The VARNUM function returns this number.
Type: Numeric

Details If no label has been assigned to the column, a blank string is returned.

Example

Obtain the label of the column NAME in the SAS table MYDATA:

```
length lab $ 40;
dsid=open('mydata','i');
if dsid then
    do;
       lab=varlabel(dsid,varnum(dsid,'name'));
       rc=close(dsid);
    end;
```

See Also

"VARNUM" on page 713

VARLEN

Returns the length of a SAS table column

Category: Variable

Syntax

length=**VARLEN**(*table-id*,*var-num*);

length
contains the length of the column.
Type: Numeric

table-id
is the identifier that was assigned when the table was opened. If *table-id* is invalid, the program halts.
Type: Numeric

var-num
is the number of the column in the SAS table. This number is adjacent to the column in the list that is produced by the CONTENTS procedure. The VARNUM function returns this number.
Type: Numeric

Example

Obtain the length of the column ADDRESS in the SAS table MYDATA:

```
dsid=open('mydata','i');
if dsid then
    do;
       namelen=varlen(dsid,varnum(dsid,'address'));
       rc=close(dsid);
    end;
```

See Also

"VARNUM" on page 713

VARLEVEL

Reports the unique values of a SAS table column

Category: Variable

Syntax

rc=**VARLEVEL**(*array-name*,*n-level*,*table-id*,*var-name*);

rc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

array-name
is the array that will contain the unique column values. This should be a character array with an element size that is large enough to hold the longest value. VARLEVEL assigns to array items the unique values of the SAS table column *var-name*.
Type: Character

n-level
is the name of the variable in which the function stores the number of unique values (or levels). This variable must be initialized to a nonmissing value before its value is set by the VARLEVEL function.

Note: This parameter is an update parameter. See "Input, Output, and Update Parameters" on page 34 for more information. △
Type: Numeric

table-id
is the identifier that was assigned when the table was opened. If *table-id* is invalid, the program halts.
Type: Numeric

var-name
is the column for which unique values are to be returned.
Type: Character

Details VARLEVEL fills the array *array-name* with the unique values of the SAS table column *var-name*.

This function returns values to the specified array. It also returns the total number of unique values in the *n-level* argument. Therefore, the second argument to this function cannot be a literal. If the number of unique values found exceeds the dimension of the array, the function returns only DIM(*array-name*) levels. That is, VARLEVEL requires the static allocation of an array that is big enough to hold all the unique values.

LVARLEVEL provides the same functionality, but it stores the unique values in an SCL list rather than an array. Because an SCL list can grow dynamically, you should consider using it rather than VARLEVEL.

Example

Get the unique formatted values for the table column X. Use ASORT to sort those values in ascending order. If NLEVELS is greater than 25, then only the first 25 values are written to the array.

```
array values {25} $ 20;
tableid=open('mylib.data','i');
nlevels=0;
rc=varlevel(values,nlevels,tableid,'x');
rc=asort(values);
do i=1 to dim(values);
   put values(i);
end;
rc=close(tableid);
```

See Also

"LVARLEVEL" on page 531

"VARNAME" on page 712

"VARSTAT" on page 714

VARLIST

Displays a dialog window that lists the columns in a SAS table, and returns the user's selections

Category: SAS Table

Syntax

selections=**VARLIST**(*table-id*,*var-type*,*num-sel*<,*message*
<,*autoclose*<,*sel-order*<,*exclude*<,*select*>>>>>);

selections
contains one or more user selections from the list, or a blank if no selection is made. Multiple selections are separated by blanks. By default, *selections* is 200 bytes long. To accomodate values longer than 200 bytes, explicitly declare *selections* with a longer length.
Type: Character

table-id
is the identifier that was assigned when the table was opened. If *table-id* is invalid, the program halts.
Type: Numeric

var-type
specifies the data type of the columns to display in the list:

'C'	The list contains only character columns.
'N'	The list contains only numeric columns.
'A'	The list contains all of the columns. (This is the default.)

Type: Character

num-sel
is the maximum number of items a user can select from the list. The default is 1. To display the list for information purposes only (no selections allowed), specify **0**. To specify an unlimited number of selections, use a value such as 9999 that is larger than the number of available selections. A user cannot make a number of selections that exceeds the number of items in the list.
Type: Numeric

message
is the text for a message to display above the selection list. The default message tells users to make up to the number of selections specified in *num-sel*.
Type: Character

autoclose
is an obsolete argument but is retained for compatibility with earlier releases. If you want to specify a value for *sel-order*, *exclude*, or *select*, then specify `''`as a placeholder for this argument.
Type: Character

sel-order
is an obsolete argument but is retained for compatibility with earlier releases. If you want to specify a value for *exclude* or *select*, then you must specify `''`as a placeholder.
Type: Character

exclude
lists one or more columns to exclude from the selection list. Separate multiple columns with at least one blank.
Type: Character

select
lists one or more columns to select for the selection list. Separate multiple columns with at least one blank.
Type: Character

Details VARLIST opens a dialog window in which a user can select columns from a SAS table. The columns specified in the VARLIST function are listed in the **`Available`** list. To make a selection, a user selects one or more columns and then presses the arrow control that points to the **`Selected`** list. The selected values move to the **`Selected`** list and are removed from the **`Available`** list. The **`Variable Details`** fields display the type, length, and description of a selected column.

For each column in the table, VARLIST uses the following steps to determine which columns to display:

- □ If one or more column names are specified in the *select* argument, VARLIST includes those columns in the selection list. Columns that are not in the *select* argument list do not appear in the selection list.
- □ If one or more column names are specified in the *exclude* argument, VARLIST excludes those columns from the selection list.
- □ If a value is specified for the *var-type* argument, VARLIST excludes all columns that are not of the specified type.

You can provide default values that will be initially selected when the column selection list is displayed. To do this, assign the column name to the *selections* variable before calling VARLIST. When the window opens, the column name appears in the **`Selected`** list.

If a user closes the dialog window without making a selection, VARLIST returns a blank value unless there was an initial value for the *selections* column before VARLIST was called.

The values for all selections can be returned in the current result list, if one is available. The current result list is a special SCL list that is automatically filled with the values selected from a selection list. To use a current result list, use the MAKELIST function to create the list, and use the CURLIST function to designate it as the current result list. The current result list must exist before you call the VARLIST function. You can use GETITEMC to retrieve these selections.

Examples

- □ Display all columns and allow the user to select only one:

```
select=varlist(dsid,'a',1);
```

- □ This next statement displays only character columns, allows two selections, uses a custom message, moves the selections to the top of the list when they are selected, and excludes the columns NAME and ADDRESS:

```
select=varlist(dsid,'c',2,'Choose a column',
               '','','name address');
```

- □ Display a dialog window that contains the character columns from an open SAS table, excluding the columns NAME and ADDRESS. Users can make two selections. The selected column names are retrieved from the current result list. LISTLEN returns the number of selections because there is only one element in the list for each selection made.

```
listid=makelist();
rc=curlist(listid);
select=varlist(dsid,'c',2,
 'Choose a column','','','name address');
n=listlen(listid);
do i=1 to n;
   varname=getitemc(listid,i);
   put varname=;
end;
```

See Also

"DATALISTC and DATALISTN" on page 326

"LISTC and LISTN" on page 518

"SHOWLIST" on page 667

VARNAME

Returns the name of a SAS table column

Category: Variable

Syntax

var-name=**VARNAME**(*table-id*,*var-num*);

var-name
contains the name of the column.
Type: Character

table-id
is the identifier that was assigned when the table was opened. If *table-id* is invalid, the program halts.
Type: Numeric

var-num
is the number of the column in the SAS table. This is the number that is adjacent to the column in the list that is produced by the CONTENTS procedure. The VARNUM function returns this number.
Type: Numeric

Example

Copy the names of the first five columns in the SAS table CITY into an SCL variable. The column names are separated by blanks.

```
length varlist $ 80;

tableid=open('city','i');
varlist=' ';
do i=1 to min(5,attrn(tableid,'nvars'));
   j=9*(i-1)+1;
   substr(varlist,j,8)=varname(tableid,i);
end;
```

```
put varlist=;
rc=close(tableid);
```

See Also

"VARNUM" on page 713

VARNUM

Returns the number of a SAS table column

Category: Variable

Syntax

var-num=**VARNUM**(*table-id*,*var-name*);

var-num
contains the position number of the column in the SAS table, or 0 if the column is not in the SAS table.
Type: Numeric

table-id
is the identifier that was assigned when the table was opened. If *table-id* is invalid, the program halts.
Type: Numeric

var-name
is the name of the column.
Type: Character

Details VARNUM returns the number of a SAS table column.

Example

Obtain the number of a column in the SAS table MYDATA, using the value of the NAME variable:

```
tableid=open('mydata','i');
if (sasname(name)) then do;
   citynum=varnum(tableid,name);
   if (citynum=0) then
      _msg_='The column is not in the table.';
   else
      _msg_='The column number is '|| citynum;
end;
else
   _msg_='Invalid SAS name--please reenter.';
```

See Also

"VARNAME" on page 712

VARSTAT

Calculates simple statistics for SAS table columns

Category: Variable

Syntax

rc=**VARSTAT**(*table-id*,*varlist-1*,*statistics*,*varlist-2*);

rc

contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

table-id

is the identifier that was assigned when the table was opened. If *table-id* is invalid, the program halts.

Type: Numeric

varlist-1

specifies one or more numeric columns for which to create the statistics.

Type: Character

statistics

specifies one or more statistics, separated by blanks, from the table below. For more information about these statistics, see the *SAS/STAT User's Guide*.

Type: Character

varlist-2

specifies one or more output variables to contain the values produced by the specified statistics. The number of output variables must equal the number of columns in *varlist-1* multiplied by the number of *statistics*.

Note: These parameters are update parameters. See "Input, Output, and Update Parameters" on page 34 for more information. △

Type: Numeric

Statistic	Description
CSS	sum of squares of a column's values, corrected for the mean
CV	coefficient of variation of a column's values
KURTOSIS	kurtosis of a column's values
MAX	largest value for a column
MEAN	mean of a nonmissing column's values

Statistic	Description
MEDIAN	median value for a column
MIN	smallest value for a column
MODE	value with the most rows for a column
N	number of rows on which calculations are based
NMISS	number of rows with missing values
NUNIQUE	number of rows having a unique value for a column
RANGE	range of values for a column
SKEWNESS	skewness of a column's values
STD	standard deviation of a column's values
STDERR	standard error of the mean
SUM	sum of nonmissing column values
USS	uncorrected sum of squares for a column
VAR	variance of a column's values

Details If more than one input column is specified with more than one statistic, then each statistic is calculated on all columns before the next statistic is calculated.

Example

Calculate the maximum, mean, and minimum values for the columns I and X from the table MY.NUMBERS:

```
tablename='my.numbers';
length imax xmax imean xmean xmin imin 8;
 /* Declare the results as numeric. */
varname='i x';
numberid=open(tablename);
if (numberid=z0) then
   do;
      _msg_='Cannot open '||tablename;
      return;
   end;
statcode=varstat(numberid,varname,
   'max mean min',imax,xmax,imean,xmean,imin,xmin);
put 'Column X';
put xmax= xmean= xmin=;
put 'Column I';
put imax= imean= imin=;
rc=close(numberid);
return;
```

See Also

"LVARLEVEL" on page 531

"VARLEVEL" on page 708

VARTYPE

Returns the data type of a SAS table column

Category: Variable

Syntax

type=**VARTYPE**(*table-id*,*var-num*);

type
contains the data type of the column:

C	character column
N	numeric column

Type: Character

table-id
is the identifier that was assigned when the table was opened. If *table-id* is invalid, the program halts.
Type: Numeric

var-num
is the position number of the column in the SAS table. This is the number that is adjacent to the column in the list that is produced by the CONTENTS procedure. VARNUM function returns this number.
Type: Numeric

Example

Place the first five numeric columns of the SAS table MYDATA into an SCL variable:

```
length varlist $ 44;
dsid=open('mydata','i');
varlist=' ';
j=0;
do i=1 to nvar(dsid) while (j<5);
if (vartype(dsid,i)='N') then
   do;
      varlist=varlist||' '||varname(dsid,i);
      j+1;
   end;
end;
rc=close(dsid);
```

See Also

"MODVAR" on page 551

"VARNUM" on page 713

WAIT

Suspends execution of the next program statement

Category: Control Flow and Utility

Syntax

CALL WAIT(*seconds*);

seconds
is the number of seconds before execution resumes.
Type: Numeric

Details You can also use WAIT to allow the screen to refresh before performing another task.

Example

Wait for 4.5 seconds between the times stored in TIME1 and TIME2, and then display those values in the LOG window:

```
time1 = datetime();
call wait( 4.5 );
time2 = datetime();
diff=putn((time2-time1),'4.1');
put time1= time2= diff=;
```

Take the last three digits before and after the decimal point from the two times: TIME1=872.229 and TIME2=876.749. Subtracting TIME2 from TIME1 results in 4.52 seconds total wait time.

```
time1=1069234872.22999
time2=1069234876.74999
diff=4.5
```

Create numeric text entry controls named START and FINISH and assign the format DATETIME20.2 to them. Enter the following program in the frame's SCL entry:

```
init:
   start.text=datetime();
   call wait(3.1);
   finish.text=datetime();
return;
```

WDEF

Resizes the active window

Category: Window

Syntax

CALL WDEF(*start-row*,*start-col*,*num-rows*,*num-cols*);

start-row
is the starting row for the window.
Type: Numeric

start-col
is the starting column for the window.
Type: Numeric

num-rows
is the number of rows for the window.
Type: Numeric

num-cols
is the number of columns for the window.
Type: Numeric

Details The active window is redefined and displayed in the area specified by WDEF. The WDEF routine performs the same function as the WDEF command. See the SAS online Help for more information about the WDEF command.

For windows that are not of type STANDARD, you must call WDEF before the window is displayed.

WDEF is frequently used to define a smaller size for a window that will be displayed along with a legend window.

Example

Use WDEF to resize the current window to occupy the bottom half of the window before calling another program. The other program is sized to display in the top half of the window. Then, return the window back to its original size.

```
call wdef(13,1,12,80);
call wregion(1,1,11,80);
call display('tophalf.program');
call wdef(1,1,24,80);
```

See Also

"WINFO" on page 721

"WREGION" on page 730

WHERE

Applies a WHERE clause to a SAS table

Category: SAS Table

Syntax

sysrc=**WHERE**(*table-id*<,*clause-1*<, . . . ,*clause-5*>>);

sysrc
contains the return code for the operation:

0	successful
>0	not successful
0<	the operation was completed, but a warning or a note was generated. If the row is locked, it is still fetched (read in), but a *sysrc* of **`_SWNOUPD`** is returned.

Type: Numeric

table-id
is the identifier that was assigned when the table was opened. If *table-id* is invalid, the program halts.
Type: Numeric

clause-1, . . . , clause-5
is the condition for the search, expressed as a WHERE clause but without the keyword WHERE. The arguments *clause-1, clause-2, clause-3, clause-4,* and *clause-5* are treated as multiple lines of the WHERE clause.

Each of the clauses, *clause-1* through *clause-5,* can contain multiple conditions separated by the AND operator. Each clause cannot exceed 200 bytes.

If the clause starts with the keyword ALSO, then the new WHERE clause is considered to be a subclause of the current WHERE clause. Specifying no clauses undoes all current temporary WHERE clauses that have been applied to the SAS table.
Type: Character

Details The WHERE function may take advantage of indexing. The syntax of the WHERE clause is the same as for the WHERE statement in base SAS software. Any WHERE clause that is applied by the WHERE function is only temporary and is considered to be a subclause of any WHERE clause that was issued when the table was opened. To apply a permanent WHERE clause to a SAS table, use the WHERE= data set option following the table name in OPEN.

The WHERE clause subsets the rows to which you have access. You must then issue a function such as FETCH or FETCHOBS to read rows from the subset. When a WHERE clause is active, FETCHOBS fetches the specified row by counting only rows that meet the WHERE condition.

To create views with more complicated WHERE clauses, use the SQL procedure or the SUBMIT CONTINUE SQL statement.

To remove only the last WHERE condition, use

```
rc=WHERE(tableid,'undo');
```

To remove all WHERE conditions, use either of the following:

```
rc=WHERE(tableid);
rc=WHERE(tableid,'clear');
```

Examples

Example 1: Applying a Compound WHERE Clause Apply a WHERE clause to the SAS table MYDATA, which subsets the table into only those rows in which column X is equal to 1 and column Z is less than 0:

```
tableid=open('mydata','i');'
rc=where(tableid,'x=1 and z<0');
```

You can separate the WHERE clause into two clauses as follows. This is equivalent to the previous example.

```
tableid=open('mydata','i');
rc=where(tableid,'x=1','and z<0');
```

Example 2: Applying a WHERE Clause in Separate SCL Statements Instead of using one WHERE clause, you can separate the WHERE clause into two statements. The following statements are equivalent to Example 1:

```
tableid=open('mydata','i');
rc=where(tableid,'x=1');
   ...more SCL statements...
rc=where(tableid,'also z<0');
```

Example 3: Using a Numeric Variable in a WHERE Clause You can pass values from SCL variables to the WHERE clause. Subset the table referenced by TABLEID, based on the value that was entered for the numeric variable SCRNUM.

```
rc=where(tableid,'num= '||put(scrnum,5.));
```

Example 4: Using a Character Variable in a WHERE Clause To subset a table based on a character value, you can use the quote function to return the quoted value. (Otherwise, you must use double quotation marks around the WHERE condition, because the quoted value itself must be enclosed in quotation marks.) Subset the table referenced by TABLEID based on the value that was entered for the character column SCRNAME. Use the QUOTE function to return the quoted value.

```
rc=where(tableid,'name= '||quote(scrname));
```

Example 5: Combining WHERE Conditions into One Statement Combine the previous two WHERE conditions into one statement:

```
rc=where(tableid,'num= '||put(scrnum,5.)||'
   and name='||quote(scrname));
```

Example 6: Determining Whether Any Rows Meet the WHERE Condition You can use the ATTRN function with the NLOBSF argument to check for rows that meet a WHERE condition, and then conditionally execute SCL code. Apply a WHERE clause to the SAS table MYDATA, which subsets the table into only those rows in which the column X is equal to 1. Use ATTRN with NLOBSF before fetching the first row of the subset.

```
tableid=open('mydata', 'i');
rc=where(tableid,'x=1');
if attrn(tableid,'nlobsf')>0 then
   rc=fetch(tableid);
```

See Also

"ATTRC and ATTRN" on page 258

"OPEN" on page 576

"FETCH" on page 389

"FETCHOBS" on page 390

"SETKEY" on page 653

WINFO

Returns information about the current window

Category: Window

Syntax

rc=**WINFO**(*info-item*<,*aux-info*>);

rc
contains the return code for the operation.
Type: Numeric

info-item
specifies either a characteristic of the window or an action. See "Values for *Info-item*" on page 722.
Type: Character

aux-info
specifies an additional argument that is required by some *info-item* actions:

pane-number (for **PANECOL**, **PANECCOL**, **PANECROW**, and **PANEROW**)
is the number of the pane to be queried. For example, in an extended table the non-scrollable section is pane 1 and the scrollable portion is pane 2.

item-id (for **PMENUGRAY**)
is the item identifier that you specified with the ID= option for the ITEM statement in PROC PMENU when you built the menu. Use the negative of the ID number to gray a selection, and use the positive ID number to ungray a selection.

item-id (for **PMENUSTATE**)
is the item identifier that you specified with the ID= option for the ITEM statement in PROC PMENU when you built the menu. Use the negative of the ID number to turn off the check mark or radio button for the selection, and use the positive ID number to turn on the check mark or radio button. Whether the menu selection gets a check mark or radio button depends on the value of the STATE= option when the menu was built.

flag (for **POPUP**)
can have the following values:

0	disables pop-up events.
1	enables pop-up events.

Type: Numeric

Values for *Info-item*

The following list explains the meaning of the return code, *rc*, for each value that you can specify for *info-item*:

BACKCOLOR
WINFO returns 1 if the device supports background colors.

BATCH
WINFO returns 1 if the application is running in batch mode.

COMMAND
WINFO returns 1 if the window has a command line.

CURSCREEN
WINFO returns the number of the SCREEN or FRAME entry in which the cursor is located.

CURSORCOL
WINFO returns the column number of the cursor position.

CURSORROW
WINFO returns the row number of the cursor position.

GRAPHICS
WINFO returns 1 if the user's output device supports graphics.

ICON
WINFO returns the number of Institute-supplied icons available under the user's host operating system. See the SAS documentation for your operating system for information about whether SAS software supports icons under your host operating system.

LOGON
WINFO returns 1 if a logon was specified when the SAS/AF application was invoked.

MAXCOL
WINFO returns the maximum number of columns to which the window can grow, excluding the command and border areas.

MAXROW
WINFO returns the maximum number of rows to which the window can grow, excluding the command and border areas.

MONO
WINFO returns 1 if the device is monochrome.

NSCREEN
WINFO returns information about the windows currently open in an application:

FSEDIT applications return the number of screens that are defined for the current FSEDIT window.

FSVIEW applications return 4 to indicate the four parts of the FSVIEW window: the column title area, the row number or ID column area, the data area, and the area above the column titles.

SAS/AF applications return 1 or 2 if you used ^^^ or ¬ ¬ ¬ to divide the entry's window into two or more frames.

NUMCOLS
WINFO returns the current number of columns in the window.

NUMROWS
WINFO returns the current number of rows in the window.

NUMXINCH
WINFO returns the number of pixels per horizontal inch of the window (valid only for FRAME entries).

NUMYINCH
WINFO returns the number of pixels per vertical inch of the window (valid only for FRAME entries).

NUMXPIXEL
WINFO returns the window width in pixels (valid only for FRAME entries).

NUMYPIXEL
WINFO returns the window height in pixels (valid only for FRAME entries).

PANECCOL
WINFO returns the column position of the cursor relative to the pane of the window in which it resides. Use *aux-info* to specify the pane.

PANECROW
WINFO returns the row position of the cursor relative to the pane of the window in which it resides. Use *aux-info* to specify the pane.

PANECOL
WINFO returns the number of columns in a specified pane. Use *aux-info* to specify the pane.

PANEROW
WINFO returns the number of rows in a specified pane. Use *aux-info* to specify the pane.

PMENU
WINFO returns 1 if a PMENU entry has been specified for this program regardless of whether the PMENU facility is currently active for the window.

PMENUGRAY
WINFO returns 0 when it grays or ungrays selections in the window's pull-down menus. Use *aux-info* to specify which selection to gray or ungray.

PMENUSTATE
WINFO returns 0 when it enables or disables the state of a selection in the window's pull-down menus. Use *aux-info* to specify the state.

POPUP
WINFO returns 0 when it enables or disables pop-up events in the window. Use the *aux-info* argument to specify whether events are to be enabled or disabled.

STARTCOL
WINFO returns the current column on which the window starts.

STARTROW
WINFO returns the current row on which the window starts.

UICON
WINFO returns the number of user-defined icons available under the user's host operating system.

XPIXCELL
WINFO returns the width (in pixels) of the font used in the SAS windowing environment.

XPIXEL
WINFO returns the horizontal location (in pixels) of the most recent mouse event.

YPIXCELL
WINFO returns the height (in pixels) of the font used in the SAS windowing environment.

YPIXEL
WINFO returns the vertical location (in pixels) of the most recent mouse event.

Example

Store the current size of the window, and then resize it back to its original size:

```
sr=winfo('startrow');
sc=winfo('startcol');
nr=winfo('numrows');
nc=winfo('numcols');
call display('top half');
call wdef(sr,sc,nr,nc);
```

See Also

"WDEF" on page 718

"WREGION" on page 730

WNAME

Specifies a name for the active window

Category: Window

Syntax

CALL WNAME(*window-name*);

window-name
is up to 80 characters for the window's name.
Type: Character

Example

Use WNAME to change the window name to "TimeData Application":

```
call wname('TimeData Application');
```

See Also

"WDEF" on page 718

"WINFO" on page 721

"WREGION" on page 730

WORD

Returns a word from a command that was issued with the command line, function keys, command processing, or a control

Category: Command

Syntax

word-text=**WORD**(*word-pos*<,*case*>);

word-text
contains the text of the word.
Type: Character

word-pos
is the position of the word to be retrieved from the command line. Specify either 1, 2, or 3 for the first, second, or third word.
Type: Numeric

case
is the type of case conversion to be performed:

'L'	converts the word to all lowercase characters
'U'	converts the word to all uppercase characters.

By default, SAS leaves all commands in the case in which they are entered.
Type: Character

Details WORD returns the first, second, or third word of the command that was issued. A *word* is the text from the current position up to the next token, such as the end of a leading number, a blank, an operator, or a semicolon.

Note: To retrieve more than three words, use NEXTWORD. △

To support custom commands in your application, you must use a CONTROL statement with either the ENTER, ALWAYS, or ALLCMDS option specified. When one of these options is specified in the CONTROL statement and when multiple commands are specified on the command line (separated by semicolons), the MAIN section is executed for each command. MAIN is executed only once if only one command is entered.

When CONTROL ALWAYS is specified, words entered on the command line that are not valid SAS commands are not flagged in error. See "CONTROL" on page 302 for information about the advantages of each CONTROL statement option before deciding which is best for your application.

WORD cannot capture windowing environment global commands, because the SCL program is not executed when a SAS windowing environment command is issued.

Examples

Example 1: Using WORD to Return the Value of the Command Suppose a user types the command **AXX BXX CXX DXX** on the command line. Use WORD to return the value of the command.

```
word1=word(1); put ''word1 is '' word1;
word2=word(2); put ''word2 is '' word2;
word3=word(3); put ''word3 is '' word3;
/* to retrieve more than three words, use NEXTWORD. */
call nextword();
word4=word(3); put ''word4 is '' word4;
```

The following output is produced:

```
word1 is AXX
word2 is BXX
word3 is CXX
word4 is DXX
```

Example 2: Using WORD to Return an Alphanumeric Value Suppose the user enters **123abc** on the command line of the frame. Use WORD to return the value of the command.

```
init:
control enter;
return;

main:
dcl char word1 word2;
word1=word(1);
word2=word(2);
put word1= word2=;
return;
```

The following output is produced:

```
word1=123 word2=abc
```

See Also

"CONTROL" on page 302

"NEXTCMD" on page 567

"NEXTWORD" on page 568

WORDTYPE

Identifies the word type of a word on the command line

Category: Command

Syntax

type=**WORDTYPE**(*word-pos*);

type
contains one of the following word types:

DATE
The word is a SAS date constant such as 25AUG98.

DATETIME
The word is a SAS datetime constant such as 25AUG98:08:15:39.30.

EOD
There are no more words on the command line (end of command).

INTEGER
The word is an integer such as 6.

LABEL
The word type is unknown to the SAS tokenizer.

NAME
The word is a SAS name such as DATA.

NUMBER
The word is a numeric constant that contains a decimal point '.', or a scientific notation 'E' for example, 6.5.

SEMI
The word is a semicolon.

SPECIAL
The word is a special operator such as '=', '+', and so on.

STRING
The word is a character string such as MYDATA.

TIME
The word is a SAS time constant such as 08:16:30.

UNKNOWN
The word type is unknown to the SAS tokenizer.
Type: Character

word-pos
is the position of the word to be retrieved from the command line. Specify either 1, 2, or 3 for the first, second, or third word.
Type: Numeric

Details WORDTYPE returns the type of the first, second, or third word that is currently on the command line. A word is the text at the current position and up to the end of a leading number or the next blank or semicolon. You can use this function with WORD.

To support custom commands in your application, you must use a CONTROL statement with either the ENTER, ALWAYS, or ALLCMDS option specified. When CONTROL ALWAYS is specified, words entered on the command line that are not valid SAS commands are not flagged in error. See “CONTROL” on page 302 for information about the advantages of each CONTROL statement option before deciding which is best for your application.

Example

Return the type of the four words that are currently on the command line:

```
w1=word(1); w1type=wordtype(1); put w1= w1type=;
w2=word(2); w2type=wordtype(2); put w2= w2type=;
w3=word(3); w3type=wordtype(3); put w3= w3type=;
CALL NEXTWORD();
w4=word(3); w4type=wordtype(4); put w4= w4type=;
```

If a user types **`ABC = 3 9`** on the command line, then this program produces the following output:

```
w1=ABC w1type=NAME
w2==w2type=SPECIAL
w3=3 w3type=INTEGER
w4=9 w4type=INTEGER
```

See Also

"CONTROL" on page 302

"NEXTWORD" on page 568

"WORD" on page 725

WOUTPUT

Manipulates the OUTPUT window

Category: Window

Syntax

sysrc=**WOUTPUT**(*action*<,*argument-1*<, . . . ,*argument-3*>>);

sysrc
contains the return code for the operation:

0	successful
≠0	not successful

Type: Numeric

action
is an action to perform, from the list below.
Type: Character

argument-1, . . . , argument-3
are additional arguments that can be specified for some of the actions, as indicated in the following list.
Type: Character

'CLEAR'
clears the OUTPUT window. There are no additional arguments for this action.

'DROPNOTE'
clears the note marked with the 'NOTE' and causes it to use its default behavior to write only the last output.

'FILE'
: writes the current contents of the OUTPUT window to an external file. This action allows two additional arguments:

  argument-1
  : is the fileref for the external file to which the window contents are to be written. This argument is required.

  argument-2
  : optionally specifies append mode. By default, each new output replaces any current contents of the external file. Specify the value **'A'** to append the output to the external file instead.

'NOTE'
: puts the pointer at the end of the current output. The NOTE action has a direct effect on the FILE, PRINT and SAVE actions. When the output is written via WOUTPUT, the pointer is reset to get the next output.

'POPOFF'
: turns off the AUTOPOP option of the OUTPUT window.

'POPON'
: turns on the AUTOPOP option of the OUTPUT window.

'PRINT'
: prints the current contents of the OUTPUT window. This action allows three additional arguments, all of which are optional:

  argument-1
  : is the name of a form to use to control printing. Use *name*.FORM or simply *name* to specify a form in the current catalog. Use *libref.catalog.name*.FORM or *libref.catalog.name* to specify a form in a different catalog. The default form is the one specified in the FORMNAME option.

  argument-2
  : is the fileref for a print file. If this argument is blank or if it is not specified, the output is sent to the default system printer.

  argument-3
  : specifies append mode. Use the value **'A'** to append the output to the current contents of the print file. If this argument is blank or if it is not specified, then each new output replaces the previous contents of the print file.

'SAVE'
: saves the current contents of the OUTPUT window in a catalog entry of type SOURCE, LOG, or OUTPUT. This action allows three additional arguments:

  argument-1
  : is the name of the entry in which the window contents are saved. The name can be a one- or three-level name. A one-level name saves the entry in SASUSER.PROFILE.*name*.OUTPUT. A three-level name saves the entry in *libref.catalog.name*.OUTPUT. You can also use a four-level name of the form *libref.catalog.entry.type*, where *type* is SOURCE, LOG, or OUTPUT. This argument is required.

  argument-2
  : is an optional description of up to 40 characters.

  argument-3
  : optionally specifies append mode. Specify the value **'A'** to append the output to the current contents of the entry. By default, each new output replaces the previous contents of the entry.

Details The list of options varies depending on the action specified.

Example

Print the OUTPUT window and append it to the file to which the fileref EXTERN has previously been assigned. Use the form specified in a previous FORMNAME option:

```
rc=woutput('print','','extern','a');
```

WREGION

Defines the boundaries for the next window that is displayed

Category: Window

Syntax

CALL WREGION(*start-row*,*start-col*,*num-rows*,*num-cols*,*options*);

start-row
is the starting row for the next window.
Type: Numeric

start-col
is the starting column for the next window.
Type: Numeric

num-rows
is the number of rows for the next window.
Type: Numeric

num-cols
is the number of columns for the next window.
Type: Numeric

options
are one or more window attributes that the window's size needs to accommodate:

''
no command line, command menu, or scroll bars

'CMDLINE'
a command area

'HSBAR'
a horizontal scroll bar

'INNERSIZE'
interior size is determined by the values of *num-rows* and *num-cols*

'PMENU'
a command menu

'VSBAR'
a vertical scroll bar
Type: Character

Details The size of the next window depends on whether it has a command area, a pmenu, and scroll bars. (The command area includes the message line.) By default, WREGION assumes that the next window will have all of these attributes. You can use *options* to change this assumption. Note that *options* does not cause the next window to have these attributes. Rather, it helps WREGION determine the correct size.

'INNERSIZE' specifies that the values of *num-rows* and *num-cols* will control the interior size of the window. Normally, the *num-rows* and *num-cols* control the exterior size of the window (including the borders).

WREGION does not affect the size of subsequent FSEDIT windows that are opened by using CALL FSEDIT. The function cannot resize windows when a SAS/AF or FSEDIT application is called with the NOBORDER option.

Examples

□ Specify a WREGION for the LEGEND window. Notice that the *options* argument to WREGION is a null string, designating that none of the display options are used in the LEGEND window. The window size is four rows long: two lines of text plus the top and bottom borders.

```
call wregion(1,1,4,80,'');
call putlegend
   (1,'This is legend line 1','yellow','none');
call putlegend
   (2,'This is legend line 2','yellow','none');
call legend
   ('mylegend window name','','white','reverse');
```

□ Execute a WREGION function and invoke a PROGRAM entry. The application window for the entry has a command line. However, because scrolling is not necessary, the HBAR and VBAR options are not specified. Only the CMDLINE option is specified.

```
call wregion(1,1,20,80,'cmdline');
call display('another.program',a,b,c);
```

See Also

"WDEF" on page 718

"WINFO" on page 721

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

SAS® Component Language: Reference, Version 8

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

ISBN 1–58025–495–0

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

1st printing, October 1999