Overview

The Data Set Model class is part of a collection of classes that combine to create the Data Form and Data Table classes. These classes work together to enable you to:

- view and edit SAS data files
- customize the functionality of these classes for your own application development needs.

Before you customize the functionality of these classes, you should understand the structure of these classes and how they communicate.

You can use the Data Set Model class by itself to access a SAS data set. To display a data set in a table editor or form editor, you must use the Data Set Data Model class.

If this class is used in conjunction with the Data Set Data Model class and an attached viewer, messages are displayed on the message line of the frame containing the viewer. Otherwise, messages are written to the LOG or MESSAGE window.

These terms are used in this class:

Relative row number

the relative position of the row within the table (ignoring any deleted rows, and taking into account any where clause or index key).

Absolute row number

The actual physical location of the row within the table (taking into account deleted rows and ignoring any WHERE clause or index key). Absolute row number is not supported by all SAS data files.

Parent:
SASHELP.FSP.OBJECT.CLASS

Class:
SASHELP.FSP.DATAST_M.CLASS
Using the Data Set Model Class

Commands for the Data Set Model Class

The following commands are supported by the Data Set Model class through the _execCmd method. Note that SCL programs should use the method that corresponds to the command; commands are provided for interactive use by end-users.

- **n**
  - Scrolls the display to the absolute row referenced by the command. If the n value is greater than the number of rows in the table, the last row in the table is displayed.
  - This command returns an error when the access method used to read the table does not support access by absolute row number or when a WHERE clause is in effect.
  - This command corresponds to the method _gotoAbsolutePath.

- **ADD**
  - Adds a new row to the table.
  - **Note:** The ADD command is not allowed when browsing a table or if the NOADD option is specified.
  - By default, all values in a new row are missing. If an initial value has been stored for the column, the value for a new row contains the initial value for the column. If no WHERE clause or key is specified, the new row becomes the current row if the engine supports this behavior.
  - This command corresponds to the method _addRow.

- **AUTOSAVE <n>**
  - Specifies how frequently the model automatically saves the table. The autosave value determines how many rows must be modified before an automatic save is performed. By default, the table is saved automatically whenever 25 rows have been modified since the last save.
  - To check the current autosave parameter value, issue the AUTOSAVE command without specifying an n value.
  - Regardless of the AUTOSAVE parameter value, you can save the table at any time by using the SAVE command.
  - This command corresponds to methods: _getAutosave, _save, and _setAutosave.

- **CREATE table REPLACE <WITH ALL | column-list>>**
  - Creates a new SAS table using some or all of the columns from the current table. The new table duplicates both the structure and contents of the current table.
  - The CREATE command will write the rows to the table in the same order they were read by the engine. For example, if a WHERE clause or _setKey method has been applied, only those rows meeting the WHERE clause or set key condition are written to the table.
  - You can select which columns are included in the new table in the following ways:
    - by listing the desired column names as command arguments
- by using the ALL argument to select all columns in the table.

You can add a list of table options following the table name. The list must be enclosed in parentheses.

By default, the CREATE command fails with an error message if the named table already exists. Specify the REPLACE option if you want the new table to replace an existing table with the same name. If you use the REPLACE option, it must come before the column name arguments.

This command corresponds to the _saveAs method.

**DELETE <row <...row-n>>**

deletes one or more rows in the table.

The DELETE command is an editing command and is not valid when browsing a table.

**CAUTION:**

Deletions cannot be recovered. You cannot recover the contents of a deleted row.

**Note:** The DELETE command is not allowed if the NODELETE option is specified.

In record-level locking, the DELETE command deletes only the currently locked row in the table.

In member-level locking, you can delete one or more rows at a time. To delete a single row, follow the DELETE command with the row number for the row to be deleted. If you are currently editing a row, issuing the DELETE command deletes that row.

To delete multiple rows, follow the DELETE command with a list of row numbers. Separate the row numbers with at least one space. For example, issue the following command to delete rows 5 and 10:

delete 5 10

To delete a range of rows, specify the first and last row numbers of the range, separated by a dash. For example, the following command deletes all rows between 5 and 10, inclusive:

delete 5-10

This command corresponds to the method _deleteRow.

**DUP <n <row>>**

copies the current row (in record-level locking). The newly copied row is immediately added to the table, and it becomes the current row if the engine supports this behavior. If you copy multiple rows simultaneously, the last row copied becomes the current row if the engine supports this behavior.

**Note:** The DUP command is not allowed when browsing a table or if the NOADD option is specified.

By default, the current row is duplicated once. To duplicate the same row again, issue the DUP command again. Alternatively, you can follow the DUP command with the desired number of copies. For example, the following command duplicates the current row three times:

dup 3

In member-level locking, the DUP command copies the specified row n times and adds the new rows to the table. If no row is specified, the current row is used.
You can select the row to copy by supplying its number as the row argument in the DUP command. To specify the row argument, you must also specify the n argument (the number of times you want the row duplicated). For example, the following command duplicates row 5 two times:

```
dup 2 5
```

This command corresponds to the method _copyRow.

**FIND** find-request

finds the next row (beginning at the current row, or at row 1 if there is no current row) that meets the specified find-request.

This command corresponds to the method _findRow.

**RFIND**

finds the next row that meets the find-request that was previously specified on a FIND command.

If the last FIND or RFIND command reached the end of the table without a match, the search begins at the beginning of the table.

This command corresponds to the method _repeatFindRow.

**SAVE**

stores all changes made to the table since the last time it was saved. See also the AUTOSAVE command.

This command corresponds to the method _save.

**SORT <ASCENDING | DESCENDING> column </options>**

sorts the table by the specified columns, and if no output table is specified, the table is sorted in place. Multiple columns can be specified. The SORT command uses the sorting program that SAS supports for your operating system. If multiple columns are specified, the table is sorted by those columns in the order in which they are specified.

The SORT command is not valid in browse mode unless you specify an output data set.

You can specify sort options to be used for the sort depending on your operating system. All sort options must be preceded by a slash (/). See the _sort method for a list of the sort options.

**UPDATE <RECORD | MEMBER>**

changes the model from browsing to editing, or changes the control level of the model when already open for editing by specifying the argument.

The UPDATE command is not allowed when the BRONLY option has been specified.

The table can be opened for editing with a control level of either RECORD or MEMBER. If you do not use either argument, the default control level used is the control level specified in the attribute window of the attached viewer or in the CNTLLEV data set option (if one is specified).

When the table is opened for editing, you can use the UPDATE command to change the current control level for the table by specifying the parameter.

The UPDATE command fails if the specified control level would cause a locking conflict. For example, you cannot specify UPDATE MEMBER if the table is open with a control level of RECORD in another window or SAS session.
This command corresponds to the method _setOpenmode.

WHERE \langle ALSO\rangle expression \rangle | \langle UNDO\rangle CLEAR

imposes one or more sets of conditions that rows in the table must meet in order to be read. Expression is any valid SAS expression involving one or more of the columns in the table. Rows that do not satisfy the specified conditions cannot be edited.

The complete set of conditions is called a temporary WHERE clause. The conditions can be modified or canceled during the lifetime of the object. In contrast, the WHERE option (specified in the ATTRIBUTE window) defines a permanent WHERE clause that cannot be changed or canceled during the FRAME session and which is not affected by WHERE commands.

The WHERE command has several forms:

WHERE expression applies the conditions specified in the expression as the new temporary WHERE clause, replacing any clause previously in effect.

WHERE ALSO expression adds the conditions specified in the expression to the existing temporary WHERE clause.

WHERE UNDO deletes the most recently added set of conditions from the temporary WHERE clause.

WHERE | WHERE CLEAR cancels the current temporary WHERE clause.

If you use the ADD or DUP command to add a new row and enter values that do not meet the WHERE conditions, the row cannot be edited once you go to a new row.

The WHERE command cannot be used in conjunction with the _setKey method.

This command corresponds to the method _setWhere.

Column Attributes

The Data Set Model class supports the column attributes shown in Table 1.

Note: You cannot change the NAME, TYPE, or LENGTH of a column.

Table 61.1 Column Attribute Items

<table>
<thead>
<tr>
<th>Item</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'NAME'</td>
<td>C</td>
<td>the name of the column: must be eight or fewer characters in length, must be a valid SAS name</td>
</tr>
<tr>
<td>'TYPE'</td>
<td>C</td>
<td>the type of the column: 'C' for character columns 'N' for numeric columns</td>
</tr>
<tr>
<td>'LENGTH'</td>
<td>N</td>
<td>the data length for the column: for character items, the maximum length is 200 for numeric items, the maximum length is 8</td>
</tr>
</tbody>
</table>
### Data Set Model Attributes

The Data Set Model class supports the following data set attributes:

<table>
<thead>
<tr>
<th>Item</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'FORMAT'</td>
<td>C</td>
<td>the format name for the column: must be appropriate for the type of the column</td>
</tr>
<tr>
<td>'INFORMAT'</td>
<td>C</td>
<td>the informat name for the column: must be appropriate for the type of the column</td>
</tr>
<tr>
<td>'LABEL'</td>
<td>C</td>
<td>the label for the column: may be a maximum of 40 characters</td>
</tr>
<tr>
<td>'INITVALUE'</td>
<td>C</td>
<td>N</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>List item</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'DATASET_NAME'</td>
<td>C</td>
<td>the name of the open table, if any</td>
</tr>
<tr>
<td>'MEMBER_TYPE'</td>
<td>C</td>
<td>the member type for the table, for example, DATA or VIEW</td>
</tr>
<tr>
<td>'ENGINE'</td>
<td>C</td>
<td>the name of the engine used for the table</td>
</tr>
<tr>
<td>'CREATE_DATE'</td>
<td>N</td>
<td>the creation date of the table as a datetime value</td>
</tr>
<tr>
<td>'MODIFY_DATE'</td>
<td>N</td>
<td>the last modified date of the table as a datetime value</td>
</tr>
<tr>
<td>'PROTECTION'</td>
<td>C</td>
<td>the level of password protection on the table: READ, WRITE, or ALTER</td>
</tr>
<tr>
<td>'DATASET_TYPE'</td>
<td>C</td>
<td>the type for the displayed table</td>
</tr>
<tr>
<td>'OPENMODE'</td>
<td>C</td>
<td>the open mode for the table: EDIT or BROWSE</td>
</tr>
<tr>
<td>'LOCK_LEVEL'</td>
<td>C</td>
<td>the locking level for the table: RECORD or MEMBER</td>
</tr>
<tr>
<td>'LOCKED_ROW'</td>
<td>N</td>
<td>the relative row number of the row that is locked. If no row is locked, a value of -1 is returned.</td>
</tr>
<tr>
<td>'LABEL'</td>
<td>C</td>
<td>the label for the table</td>
</tr>
<tr>
<td>'NUMBER_ofROWS'</td>
<td>N</td>
<td>the number of logical rows in the table, if known (those not marked for deletion). It returns -1 if unknown.</td>
</tr>
<tr>
<td>'NUMBER_ofCOLUMNS'</td>
<td>N</td>
<td>the number of columns in the table</td>
</tr>
<tr>
<td>'INDEXES'</td>
<td>N</td>
<td>the index status of the table:</td>
</tr>
<tr>
<td>List item</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>------</td>
<td>-----------------------------------------------------------------</td>
</tr>
<tr>
<td>the table is indexed</td>
<td>1</td>
<td>1 the table is indexed</td>
</tr>
<tr>
<td>0 the table is not indexed</td>
<td>0</td>
<td>0 the table is not indexed</td>
</tr>
<tr>
<td>'NUMBER_ofDeletedROWS'</td>
<td>N</td>
<td>the number of deleted rows in the table</td>
</tr>
<tr>
<td>'COMPRESSED'</td>
<td>C</td>
<td>the type of compression used for the table</td>
</tr>
<tr>
<td>'SORTED'</td>
<td>N</td>
<td>the sort status of the table:</td>
</tr>
<tr>
<td>1 the table is sorted</td>
<td>1</td>
<td>1 the table is sorted</td>
</tr>
<tr>
<td>0 the table is not sorted</td>
<td>0</td>
<td>0 the table is not sorted</td>
</tr>
<tr>
<td>'COLUMN_LIST'</td>
<td>N</td>
<td>the identifier of an SCL list that contains character items for each column in the table</td>
</tr>
</tbody>
</table>
Methods

Methods specific to the Data Set Model Class are described here. Inherited methods are described in the Object class.

Dictionary

_addRow

Adds a row to the table

Syntax

CALL SEND (object-id, '_addRow');

Details

If you modify the current row in the table, you must use the _updateRow method before calling the _addRow method.

The _addRow method is not valid when you browse a table or when the NOADD option is specified in the _setDataset method.

By default, all values in a new row are missing. If an initial value has been stored for the column (using the _setInitial method), the new row contains the initial values for the columns.

The new row is immediately added to the table. If no WHERE clause or key is specified, the new row becomes the current row if the engine supports this behavior.

The _addRow method sets SYSRC for error, note, and warning conditions.
**_copyRow**

Copies the current row

**Syntax**

CALL SEND (object-id, '_copyRow'=<, nrows>);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nrows</td>
<td>N</td>
<td>specifies the number of copies to make of the current row. The default is 1.</td>
</tr>
</tbody>
</table>

**Details**

If you modify the current row in the table, you must use the _updateRow method before calling the _copyRow method.

The _copyRow method is not valid when you browse a table or when the NOADD option is specified in the _setDataset method. The newly copied row is immediately added to the table, and it becomes the current row. If you copy multiple rows simultaneously, the last row copied becomes the current row if the engine supports this behavior.

The _copyRow method sets SYSRC for error, note, and warning conditions.

---

**_deleteRow**

Deletes the current row from the table

**Syntax**

CALL SEND (object-id, '_deleteRow');

**Details**

The _deleteRow method is not valid when you browse a table or when the NODELETE option is specified in the _setDataset method. A row must be locked before it can be deleted. Once the deletion is completed, the model no longer has a current row.

The _deleteRow method sets SYSRC for error, note, and warning conditions.

**CAUTION:**

Deletions cannot be recovered. The _deleteRow method deletes rows from the table, not just from the display of the table. You cannot recover the contents of a deleted row.
_describeColumns

Copies column attributes into a data vector

Syntax

CALL SEND (object-id, '_describeColumns', datavector-id, numcols<, startcol>);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>datavector-id</td>
<td>N</td>
<td>specifies the object identifier of an instance of the Data Set Data Vector class</td>
</tr>
<tr>
<td>numcols</td>
<td>N</td>
<td>specifies the number of columns to describe</td>
</tr>
<tr>
<td>startcol</td>
<td>N</td>
<td>specifies the column number where descriptions should start</td>
</tr>
</tbody>
</table>

Details

Column descriptions are stored in the Data Set Data Vector instance and are accessed using methods of the Data Set Data Vector class. The _describeColumns method initializes the Data Set Data Vector instance with the format, informat, label, name, and type for the specified columns. See the Data Set Data Vector class for more information on how to access or override these attributes.

_execcmd

Processes a command

Syntax

CALL SEND (object-id, '_execcmd');

Details

Use the _execcmd method when you want the model to process commands. Note that the command is not passed as an argument. Rather, it processes commands that are sent to the object. Commands are processed in the following order:

1. If the command was issued while in a data form object or data table object, the viewer tries to process the command.
2. If the viewer does not recognize the command, the command is sent to the data set data model object.
3. If the data set data model object does not recognize the command, the command is sent to the data set model object.
If the data set model object does not recognize the command, the command is returned to the frame.

For an example of using the _execCmd method, see the _execCmd method in the Data Set Data Model class.

_fetchRow

Reads a row from the table

Syntax
CALL SEND (object-id, '_fetchRow', row-num<, eod>);

Argument Type Description
---
row-num N specifies the relative row number to be read. This row becomes the current row.
eod N returns whether the end of the data has been reached
   1 end of data has been reached
   0 more records are to be read

Details
If you modify the current row in the table, you must use the _updateRow method before calling the _fetchRow method.

The _fetchRow method uses the last Data Set Data Vector. You cannot call the _fetchRow method until you pass a Data Set Data Vector instance to another method of this object, such as the _getRow method.

The _fetchRow method sets SYSRC for error, note, and warning conditions.

Example

In the following example, the Data Set Model class is being used as a stand-alone model, that is, the model class is not being used with a data form or data table object. This example is written as a stand-alone SCL entry.

Note: You would not use this same code with a data form or data table object because in that case, the model has already been initialized for you.

```
length name $ 8;

Load the Data Set Model.

INIT:
```
```plaintext
dataid = instance( loadclass
    ( 'sashelp.fsp.datast_m.class' ) );
dsdvec = instance( loadclass
    ( 'sashelp.fsp.dsdvec.class' ) );

Set the data set.

call send( dataid, '_setDataset',
         'sasuser.class' );
call send( dataid, '_getNumberOfColumns',
        ncols );
call send( dsdvec, '_setNumberOfColumns',
        ncols, dataid );

Create the WHERE list to subset the dataset. Select all males who are taller than 5 feet.

wlst = makelist();
wlst = insertc( wlst, 'sex="M"', -1);
wlst = insertc( wlst, 'and height > 60', -1);
call send( dataid, '_setWhere', wlst );
wlst = dellist( wlst );

Get the first row from the table. Initialize the instance of the data set model object with the data set vector object.

row = 1;
call send( dataid, '_getRow',
        dsdvec, row, eod );

Loop through all of the rows until no more meet the WHERE criteria.

do while( not eod );

Get the values for the NAME and WEIGHT column and use the values in a PUT statement.

call send( dataid, '_getColumnText',
        'NAME', name );
call send( dataid, '_getColumnValue',
        'height', height );
put name= height=;
```
Increment the row by one. Determine whether another row exists that meets the selection criteria. The _fetchRow method is used here instead of _getRow method. This approach works because the data set model object was initialized with the dataset data vector object. (See the fourth statement in this example, which uses the LOADCLASS and INSTANCE functions.)

```
row + 1;
call send( dataid, '_fetchRow',
    row, eod );
end;
```

If the row is still set to 1, then no rows meet the WHERE criteria.

```
if ( row = 1 ) then
    put 'No rows meet the WHERE clause criteria';
return;
```

Terminate the classes that were instantiated.

```
TERM:
call send( dsdvec, '_term' );
call send( dataid, '_term' );
return;
```

### _findRow

Returns the number of the row that meets the find request

#### Syntax

```
CALL SEND (object-id, '_findRow', find-request<, start-row>);
```
### Argument Type Description

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>find-request</td>
<td>N</td>
<td>specifies the identifier of an SCL list that contains the find request</td>
</tr>
<tr>
<td>start-row</td>
<td>N</td>
<td>specifies the row to start the search on. The search begins with start-row. If the find is successful, start-row returns the row number of the match; otherwise, it returns -1. By default, if no start-row is defined, the search begins with the current row plus 1. If there is no current row, the search begins with row 1.</td>
</tr>
</tbody>
</table>

### Details

If you modify the current row in the table, you must use the _updateRow method before calling the _findRow method.

The find request should contain one or more character list items. Each item should use standard WHERE clause syntax and is handled as an additional request.

If the end of the table is reached without a match, the last row of the table becomes the current row. If the last _findRow or _repeatFindRow reached the end without a match, the search begins at the beginning of the table.

### Example

In the following example, the Data Set Model class is being used as a stand-alone model, that is, the model class is not being used with a data form or data table object.

This example uses the _findRow method and the _repeatFindRow method. The example assumes you have created a FRAME entry with two push buttons named BUTTON1 and BUTTON2.

```sas
length charval $ 15;

Load the Data Set Model. In this example, the data set model object is being used as a stand-alone object. Therefore, if you are using SCL similar to this with a data form object or data table object, you should omit the LOADCLASS and INSTANCE functions.

INIT:

datcl=loadclass('sashelp.fsp.datast_m.class');
datid=instance(datcl);

Set the data set.

    call send(datid, '_set_dataset_',
              'sasuser.crime');

return;
```
When the first push button is pressed, find the row that has STATE =10 or STATEN="DELAWARE".

BUTTON1:
row=1;
list=makelist();
list=insertc(list,'STATE=10');
list=insertc(list,'or upcase(staten)=
"DELAWARE"','-1');
call send(datid,'_find_row_',list,row);
call send(datid,'_get_column_text_', 'staten',charval);
call send(datid,'_get_column_value_', 'state',numval);
put row= charval= numval=;
list=dellist(list);
return;

When the second push button is pressed, repeat the find request.

BUTTON2:
row+1;
call send(datid, '_repeatFindRow', row);
put row=;
return;

Terminate the classes that were instantiated.

TERM:
call send(datid,'_term_');
return;

See Also
_repeatFindRow.

__getAutosave

Returns the current autosave setting

Syntax
CALL SEND (object-id, '_getAutosave', autosave);
**_getColumnAttribute**

Returns a single attribute for a column

**Syntax**

CALL SEND (object-id, '_getColumnAttribute', col-name, attr-name, attr-value);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>col-name</td>
<td>C</td>
<td>specifies the name of the column from which to get the attribute</td>
</tr>
<tr>
<td>attr-name</td>
<td>C</td>
<td>specifies the name of the column attribute</td>
</tr>
<tr>
<td>attr-value</td>
<td>C</td>
<td>N</td>
</tr>
</tbody>
</table>

**Details**

The type of attr-value must match the type of the attribute to be returned. For more information about the appropriate attribute names, see “Column Attributes” on page 1057.

To return more than one column attribute, use the _getColumnAttributes method.

**Example**

In the following example, the Data Set Model class is being used as a stand-alone model, that is, the model class is not being used with a data form or data table object.

This example is written as a stand-alone SCL entry:

```scl
length cvalue $10;
```

Load the Data Set Model. In this example, the data set model object is being used as a stand-alone object. Therefore, if you are using SCL similar to this with a data form object or data table object, you should omit the LOADCLASS and INSTANCE functions.

```scl
INIT:
   datcl=loadclass('sashelp.fsp.
```
datast_m.class’);
datid=instance(datcl);

Set the data set.

    call send(datid, '_set_dataset_',
                'sasuser.class');

Get the type of the SEX column.

    call send(datid, '_get_column_attribute_',
               'sex', 'type', cvalue);
    put 'The type of the SEX column=' cvalue;

Get the length of the AGE column.

    call send(datid, '_get_column_attribute_',
               'age', 'length', nvalue);
    put 'The length of the AGE column=' nvalue;
    return;

Terminate the classes that were instantiated.

TERM:
    call send(datid,'_term_');
    return;

_getColumnAttributes

Returns the attributes for a column

Syntax

CALL SEND (object-id, '_getColumnAttributes', list-id);
<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list-id</td>
<td>N</td>
<td>specifies the identifier of an SCL list to contain the current column attributes</td>
</tr>
</tbody>
</table>

**Details**

This list must contain at least a single character item with an item name of NAME, the value of which must be a valid column name.

By default, all attributes are returned. To return only certain column attributes, include the appropriate named items in your list. See “Column Attributes” on page 1057 for more information on specific column attributes.

To return a single column attribute, use the `_getColumnAttribute` method.

**Example**

In the following example, the Data Set Model class is being used as a stand-alone model, that is, the model class is not being used with a data form or data table object.

This example is written as a stand-alone SCL entry:

**Example Code 61.1**

Load the Data Set Model. In this example, the data set model object is being used as a stand-alone object. Therefore, if you are using SCL similar to this with a data form object or data table object, you should omit the LOADCLASS and INSTANCE functions.

```sas
INIT:
   datcl=loadclass('sashelp.fsp.
                   dataset_m.class');
   datid=instance(datcl);

   call send(datid, '_set_dataset_',
             'sasuser.class');

   gattr=makelist();
   gattr=setnitemc(gattr, 'sex', 'name');
   call send(datid, '_get_column_attributes_',
             gattr);
   call putlist(gattr, 'All Attributes for SEX=’, 0);

   call send(datid, '_set_dataset_',
             'sasuser.class');

   gattr=makelist();
   gattr=setnitemc(gattr, 'height', 'format');
   call send(datid, '_set_dataset_',
             'sasuser.class');
```

Set the format for column HEIGHT.
clearlist(gattr);
gattr=setnitemc(gattr,'height','name');
gattr=setnitemc(gattr,'4.1','format');
call send(datid,'_set_column_attributes_',
gattr);

clearlist(gattr);
gattr=setnitemc(gattr,'height','name');
gattr=setnitemc(gattr,' ','format');
call send(datid,'_get_column_attributes_',
gattr);
call putlist(gattr,'Format of
  Column HEIGHT=',0);
dellist(gattr);
return;

Terminate the classes that were instantiated.

TERM:
call send(datid,'_term_');
return;

__get_ColumnFormattedData

Returns the formatted column data

Syntax
CALL SEND (object-id, '_get_ColumnFormattedData', col-name, data);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>col-name</td>
<td>C</td>
<td>specifies the name of the column for which the formatted data is requested</td>
</tr>
<tr>
<td>data</td>
<td>C</td>
<td>returns the formatted data of the requested column</td>
</tr>
</tbody>
</table>

Details
The _get_ColumnFormattedData method provides an easy way to obtain formatted data for a column in the currently locked row. For example, using the _getColumnValue
method on a date column would return the internal date value, but using _getColumnFormattedData would return the formatted date value.

The _getColumnFormattedData method uses the last Data Set Data Vector. You cannot call the _getColumnFormattedData method until you pass a Data Set Data Vector instance to another method of this object, such as the _getRow method.

---

**_getColumnNumber**

**Returns the column number for a column**

**Syntax**

CALL SEND (object-id, '_getColumnNumber', col-name, col-num);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>col-name</td>
<td>C</td>
<td>specifies the name of the column for which the number is requested</td>
</tr>
<tr>
<td>col-num</td>
<td>N</td>
<td>returns the number for the named column or -1 if the column does not exist</td>
</tr>
</tbody>
</table>

**Details**

The _getColumnNumber method returns the requested position within a SAS file.

---

**_getColumnText**

**Returns the text for a character column in the current row**

**Syntax**

CALL SEND (object-id, '_getColumnText', col-name, text);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>col-name</td>
<td>C</td>
<td>specifies the name of the column for which the text is requested</td>
</tr>
<tr>
<td>text</td>
<td>C</td>
<td>returns the text of the requested column</td>
</tr>
</tbody>
</table>

**Details**

The _getColumnText method uses the last Data Set Data Vector. You cannot call the _getColumnText method until you pass a Data Set Data Vector instance to another method of this object, such as the _getRow method.
The _getColumnText method sets SYSRC for error, note, and warning conditions.

### _getColumnValue

**Returns the value for a numeric column in the current row**

#### Syntax

CALL SEND (object-id, '_getColumnValue', col-name, value);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>col-name</td>
<td>C</td>
<td>specifies the name of the column for which the value is requested</td>
</tr>
<tr>
<td>value</td>
<td>N</td>
<td>returns the value for the column</td>
</tr>
</tbody>
</table>

#### Details

The _getColumnValue method uses the last Data Set Data Vector. You cannot call the _getColumnValue method until you pass a Data Set Data Vector instance to another method of this object, such as the _getRow method.

The _getColumnValue method sets SYSRC for error, note, and warning conditions.

### _getCurrentRowNumber

**Returns the relative row number of the current row**

#### Syntax

CALL SEND (object-id, '_getCurrentRowNumber', row-num);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>row-num</td>
<td>N</td>
<td>returns the relative row number of the current (locked) row. It returns -1 if no row is current.</td>
</tr>
</tbody>
</table>

### _getDatasetAttributes

**Returns the attributes for the table**
Syntax

CALL SEND (object-id, '_getDatasetAttributes', list);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>N</td>
<td>specifies the identifier of an SCL list to contain the attributes</td>
</tr>
</tbody>
</table>

Details

To query all table attributes, the list should be empty. To query specific attributes, the list should contain named items for attributes of interest. See “Data Set Model Attributes” on page 1058 for details on specific attributes.

The _getDatasetAttributes method sets SYSRC for error, note, and warning conditions.

Example

In the following example, the Data Set Model class is being used as a stand-alone model, that is, the model class is not being used with a data form or data table object.

This example assumes you have created a FRAME entry with a push button object named BUTTON1:

```
Load the Data Set Model. In this example, the data set model object is being used as a stand-alone object. Therefore, if you are using SCL similar to this with a data form object or data table object, you should omit the LOADCLASS and INSTANCE functions.

INIT:
   datcl=loadclass('sashelp.fsp.
   dataset_m.class');
   datid=instance(datcl);

Set the data set.

   call send(datid, '_set_dataset_',
   'sasuser.class');
   return;

Get all of the data set attributes.

BUTTON1:
   gattr=makelist();
   call send(datid, '_get_dataset_attributes_',
```
Data Set Model Class

_getNumberOfColumns

Returns the number of columns in the table

_getDatasetName

Returns the name of the data set opened with the _setDataset method

Syntax

CALL SEND (object-id, '_getDatasetName', name);

Argument | Type | Description
---|---|---
name | C | returns the name of the data set opened with the _setDataset method, including any data set options passed to _setDataset as part of the data set name. It returns a blank if no data set is open.

See Also

_setDataset.
Syntax

CALL SEND (object-id, '_getNumberOfColumns', numcols);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>numcols</td>
<td>N</td>
<td>returns the number of columns in the table</td>
</tr>
</tbody>
</table>

Details

The value returned by this method can be passed to the _setNumberOfColumns method of the Data Set Data Vector class to initialize the data vector.

__getRow

Reads a row from the table into the data vector class

Syntax

CALL SEND (object-id, '_getRow', datavector-id, row-num<, eod>);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>datavector-id</td>
<td>N</td>
<td>specifies the object identifier of an instance of the Data Set Data Vector class. The values for the specified row are stored in the data vector and can be accessed using Data Set Data Vector class methods.</td>
</tr>
<tr>
<td>row-num</td>
<td>N</td>
<td>specifies the number of the relative row to be read. It becomes the current row.</td>
</tr>
<tr>
<td>eod</td>
<td>N</td>
<td>returns whether the end of the data has been reached</td>
</tr>
<tr>
<td>1</td>
<td>end of data has been reached</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>more records are to be read</td>
<td></td>
</tr>
</tbody>
</table>

Details

If you modify the current row in the table, you must use the _updateRow method before calling the _getRow method.

After calling _getRow, you can use the _getColumnText and _getColumnValue methods of the Data Set Model class or the _getText and _getValue methods of the Data Set Data Vector class to retrieve the values read from the table.

The _getRow method sets SYSRC for error, note, and warning conditions.
_getRowNumber

Returns the absolute row number for the current row or the specified relative row

**Syntax**

CALL SEND (object-id, '_getRowNumber', abs-row<, rel-row>);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs-row</td>
<td>N</td>
<td>returns the absolute row number for the current row, or the row number of rel-row, if specified. It returns -1 if the absolute row number is not available or there is no current row.</td>
</tr>
<tr>
<td>rel-row</td>
<td>N</td>
<td>specifies the relative row number for which the absolute row number is returned. Note that specifying rel-row changes the current row to the row referenced by rel-row.</td>
</tr>
</tbody>
</table>

**Details**

The _getRowNumber method uses the last Data Set Data Vector. You cannot call the _getRowNumber method until you pass a Data Set Data Vector instance to another method of this object, such as the _getRow method.

The _getRowNumber method sets SYSRC for error, note, and warning conditions.

_getWhere

Returns the active where clause

**Syntax**

CALL SEND (object-id, '_getWhere', list);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>N</td>
<td>specifies the identifier of an SCL list to contain the active WHERE clause, if any. If no WHERE clause is active, then the list is empty.</td>
</tr>
</tbody>
</table>

_gotoAbsoluteRow

Goes to the specified row
Syntax
CALL SEND (object-id, '_gotoAbsoluteRow', abs-row<, rel-row>);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs-row</td>
<td>N</td>
<td>specifies the absolute row number to be read. If the abs-row value is greater than the number of rows in the table, the last row in the table becomes the current row.</td>
</tr>
<tr>
<td>rel-row</td>
<td>N</td>
<td>returns the relative row number for the row that is now current.</td>
</tr>
</tbody>
</table>

Details
If you modify the current row in the table, you must use the _updateRow method before calling the _gotoAbsoluteRow method.

The _gotoAbsoluteRow method returns an error when the access engine used to read the table does not support access by absolute row number or when a permanent or temporary WHERE clause is in effect.

The _gotoAbsoluteRow method uses the last Data Set Data Vector. You cannot call the _gotoAbsoluteRow method until you pass a Data Set Data Vector instance to another method of this object, such as the _getRow method.

The _gotoAbsoluteRow method sets SYSRC for error, note, and warning conditions.

_keyCount

Returns the number of rows that meet the current index key

Syntax
CALL SEND (object-id, '_keyCount', rc, count);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rc</td>
<td>N</td>
<td>returns whether the count was successful</td>
</tr>
<tr>
<td>count</td>
<td>N</td>
<td>returns the number of rows in the data set that match the current index key that is specified with the _setKey method</td>
</tr>
</tbody>
</table>

Details
The _keyCount method returns the number of rows that meet the criteria specified by the index key column. The index key column is specified with the last _setKey method used on the table. After the _keyCount method executes, the table is positioned at the
first row that meets the criteria defined by the last _setKey method. Use the _getRow
or _fetchRow method to read the row.

The _keyCount method sets SYSRC for error, note, and warning conditions.

CAUTION:
Using the _keyCount method with composite keys may show a larger number of rows
matching the search criteria than you expect. Using a composite key with the _setKey
method operates the same way as the _where method 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 were connected by WHERE conditions that use 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 together to form the index key. The
number returned by the _keyCount method 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 may
meet the search condition that you did not expect, such as SEX of M and AGE of 11,
because the string M11 is considered greater than the string F13. △

See Also

_setKey.

_repeaFindRow

Finds the next row that meets the last find request

Syntax

CALL SEND (object-id, '_repeatFindRow'<, start-row>);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>start-row</td>
<td>N</td>
<td>specifies the row to start searching from, including a search of that row. The start-row argument returns the next matching row or -1 if no match.</td>
</tr>
</tbody>
</table>

Details

If you modify the current row in the table, you must use the _updateRow method before
calling the _repeatFindRow method.

By default, if no start-row is defined, the search begins with the current row plus 1.
If the end of the table is reached without a match, the last row of the table becomes
the current row. If the last _findRow or _repeatFindRow reached the end without a
match, the search begins at the beginning of the table.

The _repeatFindRow method uses the find request specified by the last _findRow
method call.
See Also

_findRow

__save

Saves the table

Syntax

CALL SEND (object-id, '_save');

Details

You must be in edit mode to use the _save method. When the table is saved, you no longer have a current row. Methods requiring a current row, such as the _copyRow and _deleteRow methods, fail if a new current row is not established after the save by using the _getRow or _fetchRow method. The _save method sets SYSRC for error, note, and warning conditions.

__saveAs

Saves the table under a new name

Syntax

CALL SEND (object-id, '_saveAs', name<, replace<, col-1,..., col-n>>);

Argument     Type     Description
------------- ------- -------------------------------
name          C        specifies the new name to use when saving the table
replace       C        specifies whether to replace an existing table name:
                  'N'      does not replace the table (default)
                  'Y'      replaces the table if it exists
col           C        specifies one or more column (variable) names separated by commas that specify the columns to be written to the new table. If none are specified, all are written to the new table.

Details

The _saveAs method will write the rows to the table in the same order they were read by the engine. For example, if a WHERE clause or _setKey method has been applied, only those rows meeting the WHERE clause or set key condition are written to the table.
The _saveAs method sets SYSRC for error, note, and warning conditions.

**Example**

In the following example, the Data Set Model class is being used as a stand-alone model, that is, the model class is not being used with a data form or data table object.

This example is written as a stand-alone SCL entry:

```
INIT:
    datcl=loadclass('sashelp.fsp.datast_m.
                    class');
    datid=instance(datcl);

    call send(datid,'_set_dataset_','sasuser.class');

    call send(datid,'_save_as_','work.test','y',
              'name','age','sex');
    return;

TERM:
    call send(datid,'_term_');
    return;
```

_setsAutosave

Sets a new autosave value
Syntax

CALL SEND (object-id, '_setAutosave', autosave);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>autosave</td>
<td>N</td>
<td>specifies the new autosave value for the table. This value must be greater than 0.</td>
</tr>
</tbody>
</table>

Details

The _setAutosave method specifies how frequently the model automatically saves the table. The autosave value determines how many rows must be changed before an automatic save is performed. By default, the table is saved automatically whenever 25 rows have been modified since the last save.

To check the current autosave parameter value, use the _getAutosave method.

Regardless of the autosave value, you can save the table at any time by using the _save method.

When the table is saved due to the autosave setting, you no longer have a current row. Methods requiring a current row, such as the _copyRow and _deleteRow methods, fail if a new current row is not established after the save by using the _getRow or _fetchRow method.

_setColumnAttribute

Sets an attribute for a column

Syntax

CALL SEND (object-id, '_setColumnAttribute', col-name, attr-name, attr-value);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>col-name</td>
<td>C</td>
<td>specifies the name of the column in which to set the attribute</td>
</tr>
<tr>
<td>attr-name</td>
<td>C</td>
<td>specifies the name of the column attribute to be set</td>
</tr>
<tr>
<td>attr-value</td>
<td>C</td>
<td>N</td>
</tr>
</tbody>
</table>

Details

The type of the attribute value must match the type of the attribute you specify. For more information about the appropriate attribute names, see “Column Attributes” on page 1057.

To specify more than one column attribute, use the _setColumnAttributes" _setColumnAttributes "on page 1083 method. Note: You cannot change the NAME, TYPE, or LENGTH of a column.
_setColumnAttributes

Sets the attributes for a column

Syntax

CALL SEND (object-id, '_setColumnAttributes', list-id);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list-id</td>
<td>N</td>
<td>specifies the identifier of an SCL list that contains the current column attributes</td>
</tr>
</tbody>
</table>

Details

This list must contain at least a single character item with an item name of NAME. This value is required and must be a valid column name. See “Column Attributes” on page 1057 for more information on specific column attributes.

To set a single column attribute, use the _setColumnAttribute method.

Note: You cannot change the NAME, TYPE, or LENGTH of a column.

_setColumnText

Sets the text for a character column in the current row

Syntax

CALL SEND (object-id, '_setColumnText', col-name, text);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>col-name</td>
<td>C</td>
<td>specifies the name of the character column in which to set the text</td>
</tr>
<tr>
<td>text</td>
<td>C</td>
<td>specifies the new text to assign to the character column in the current row</td>
</tr>
</tbody>
</table>

Details

The _setColumnText method uses the last Data Set Data Vector. You cannot call the _setColumnText method until you pass a Data Set Data Vector instance to another method of this object, such as the _getRow method.

The _setColumnText method sets SYSRC for error, note, and warning conditions.
_setColumnValue

Sets the value for a numeric column in the current row

Syntax

CALL SEND (object-id, '_setColumnValue', col-name, value);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>col-name</td>
<td>C</td>
<td>specifies the name of the column in which to set the value</td>
</tr>
<tr>
<td>value</td>
<td>N</td>
<td>specifies the new value to assign to the numeric column in the current row</td>
</tr>
</tbody>
</table>

Details

The _setColumnValue method uses the last Data Set Data Vector. You cannot call the _setColumnValue method until you pass a Data Set Data Vector instance to another method of this object, such as the _getRow method.

The _setColumnValue method sets SYSRC for error, note, and warning conditions.

_setDataset

Sets the data set being accessed by the object

Syntax

CALL SEND (object-id, '_setDataset', name<, mode><, locking><, option-1,...option-n>);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>C</td>
<td>specifies the name of the data set. The name may include data set options. See SAS Language Reference, Version 6, First Edition for more information.</td>
</tr>
<tr>
<td>mode</td>
<td>C</td>
<td>specifies the open mode for the new table: 'BROWSE' or 'EDIT'. The default is 'BROWSE'.</td>
</tr>
<tr>
<td>locking</td>
<td>C</td>
<td>specifies the locking level: 'RECORD' or 'MEMBER'. The default is 'RECORD'. For more information, see &quot;Control Levels&quot; in SAS Component Language Reference.</td>
</tr>
<tr>
<td>options</td>
<td>C</td>
<td>specifies any additional _setDataset method options: 'BRONLY' prevents editing of the table</td>
</tr>
</tbody>
</table>
Data Set Model Class

### Argument Type Description

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'NOADD'</td>
<td></td>
<td>prevents the user from adding new rows to the table</td>
</tr>
<tr>
<td>'NODELETE'</td>
<td></td>
<td>prevents the user from deleting rows in the table</td>
</tr>
</tbody>
</table>

### Details

If you modify the current row in the table, you must use the `_updateRow` method before calling the `_setDataset` method.

The `_setDataset` method sets SYSRC for error, note, and warning conditions.

**Note**: Passing a value of "as the name of the data set will close the current data set.

### Example

In the following example, the Data Set Model class is being used as a stand-alone model, that is, the model class is not being used with a data form or data table object.

This example is written as a stand-alone SCL entry:

Initialize NAME.

```sas
length name $8;
```

Load the classes.

```sas
INIT: 
  datcl = loadclass('sashelp.fsp.datast_m. 
                    class');
  datid = instance(datcl);
  dsdvcl = loadclass('sashelp.fsp.dsdvec. 
                    class');
  dsdvec = instance(dsdvcl);
```

Set the data set.

```sas
  call send(datid, '_setDataset', 
            'sasuser.class', 
            'EDIT', 'RECORD', 
            'NOADD', 'NODELETE');
```

Set up the data set data vector for later reference.

```sas
  call send(datid, '_getNumberOfColumns',
```
Sets the initial values to be used for any newly added row

**Syntax**

CALL SEND (object-id, '_setInitial'<, 'CLEAR'>);

**Details**

The _setInitial method sets the column values in the current row as the initial column values to be used for any added rows. If you specify the optional 'CLEAR' parameter, it restores the initial values for all columns to missing values.

Sets an index key for retrieving the rows in the table
Data Set Model Class

Syntax

CALL SEND (object-id, '_setKey', rc<, keyname<, condition<, scroll<, val-list>);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rc</td>
<td>N</td>
<td>returns whether the key was successfully applied</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 key was successfully applied</td>
</tr>
<tr>
<td></td>
<td></td>
<td>not 0 otherwise</td>
</tr>
<tr>
<td>keyname</td>
<td>C</td>
<td>specifies the name of the key or index to use on the table. The keyname may specify a single or a composite index.</td>
</tr>
<tr>
<td>condition</td>
<td>C</td>
<td>specifies the condition to use when comparing the key value:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'EQ' equal to</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'GE' greater than or equal to</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'GT' greater than</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'LE' less than or equal to</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'LT' less than</td>
</tr>
<tr>
<td>scroll</td>
<td>C</td>
<td>specifies whether observations can be retrieved in random order:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'SCROLL' observations can be retrieved in random order (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'NOSCROLL' observations can only be retrieved sequentially</td>
</tr>
<tr>
<td>val-list</td>
<td>N</td>
<td>specifies the identifier of an SCL list that contains values to use in the key. The item name should reflect the appropriate column name, and the item value should be the value that the key value is compared against.</td>
</tr>
</tbody>
</table>

Details

If you modify the current row in the table, you must use the _updateRow method before calling the _setKey method.

The _setKey method 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 table rows by evaluating the value of the columns against the key value in the rows.

Using a composite key with _setKey operates the same way as the _where method 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 were connected by WHERE conditions that use 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 together to form the index key. The number returned by the _keyCount method 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 may meet the search condition that you did not expect, such as SEX of M and AGE of 11, because the string M11 is considered greater than the string F13.
Once an active key is set through the `_setKey` method, it remains active until the following conditions are met:

- the table is closed
- another key is set
- the current setting is cleared by passing the `rc` argument alone to the `_setKey` method.

The table is automatically positioned at the row that meets the specified criteria. Use the `_getRow` or `_fetchRow` method to read the row.

The `_setKey` method cannot be used in conjunction with a WHERE clause.

The `_setKey` method sets SYSRC for error, note, and warning conditions.

**Example**

In the following example, the Data Set Model class is being used as a stand-alone model, that is, the model class is not being used with a data form or data table object.

This example creates an index on the STATE variable in the table SASUSER.CRIME. It subsets on STATE values less than 20. In this example, COUNT returns 15.

```sas
CREATE THE INDEX ON STATE. IN THIS EXAMPLE, THE DATA SET MODEL OBJECT IS BEING USED AS A STAND-ALONE OBJECT. THEREFORE, IF YOU ARE USING SCL SIMILAR TO THIS WITH A DATA FORM OBJECT OR DATA TABLE OBJECT, YOU SHOULD OMIT THE LOADCLASS AND INSTANCE FUNCTIONS.

INIT:
  dsid=open('sasuser.crime', 'v');
  icreate(dsid, 'state', 'state');
  close(dsid);

  datcl=loadclass('sashelp.fsp.
                  dataset_m.class');
  datid=instance(datcl);

SET THE DATA SET.

  call send(datid,'_set_dataset_','sasuser.crime');

SUBSET ON STATE VALUES LESS THAN 20.

  list=makelist();
  list=setnitemn(list, 20, 'state');
  call send(datid,'_set_key_','rc','state','LT','scroll',list);  
  call send(datid,'_key_count_','rc',count);
  put count=
  return;
```
Terminate the classes that were instantiated.

```
TERM:
    call send(datid,'_term_');
    dellist(list);
    return;
```

### _setMsg

Specifies the text of the message

**Syntax**

CALL SEND (object-id, '_setMsg', msg_text);

**Argument**  | **Type** | **Description**  
--- | --- | ---  
msg_text | C | specifies the text of the message to be displayed.

**Details**

The message will be displayed in the SAS log or the message window.

### _setOpenmode

Changes the open mode and lock mode for the table

**Syntax**

CALL SEND (object-id, '_setOpenmode', mode<, locking>);

**Argument**  | **Type** | **Description**  
--- | --- | ---  
mode | C | specifies the new open mode for the table: ‘EDIT’ or ‘BROWSE’. You cannot set the open mode to ‘EDIT’ if ‘BRONLY’ has been specified for _setDataset.

locking | C | specifies the new locking mode for the table: ‘MEMBER’ or ‘RECORD’. The default is ‘RECORD’.

**Details**

The _setOpenmode method sets SYSRC for error, note, and warning conditions.
**_setRow**

Sets the values in the current row from data vector object

---

**Syntax**

CALL SEND (object-id, '_setRow', datavector-id);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>datavector-id</td>
<td>N</td>
<td>specifies the object identifier for an instance of the Data Set Data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Vector class that contains the values to set in the row</td>
</tr>
</tbody>
</table>

**Details**

You must follow a _setRow call with an _updateRow to write the row to the table.

Before you call the _setRow method, you should call the _setText or _setValue methods of the Data Set Data Vector class to set the column values to the desired values.

The _setRow method sets SYSRC for error, note, and warning conditions.

---

**_setWhere**

Sets a WHERE clause on the table

---

**Syntax**

CALL SEND (object-id, '_setWhere', where-list);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>where-list</td>
<td>N</td>
<td>specifies the identifier of an SCL list that contains the WHERE clause(s)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>to apply to the table</td>
</tr>
</tbody>
</table>

**Details**

If you modify the current row in the table, you must use the _updateRow method before calling the _setWhere method.

The _setWhere method imposes one or more sets of conditions that rows in the table must meet. Rows that do not satisfy the specified conditions cannot be edited.

If you use the _addRow method to add a new row and the initial values for the row do not meet the WHERE conditions, the row cannot be fetched while the WHERE clause is in effect.

If you augment a WHERE clause, the SCL list containing the augmentation must begin with ALSO.

The _setWhere method cannot be used with the _setKey method.
The _setWhere method sets SYSRC for error, note, and warning conditions.

**Example**

In the following example, the Data Set Model class is being used as a stand-alone model, that is, the model class is not being used with a data form or data table object.

This example assumes you have created a FRAME entry with a push button object named BUTTON1:

```
Load the Data Set Model. In this example, the data set model object is being used as a stand-alone object. Therefore, if you are using SCL similar to this with a data form object or data table object, you should omit the LOADCLASS and INSTANCE functions.

INIT:
   datcl=loadclass('sashelp.fsp.datast_m.
                  class');
   datid=instance(datcl);

Set the data set.

   call send(datid, '_set_dataset_','sasuser.class');
   return;

Apply the WHERE clause

BUTTON1:
   listid=makelist();
   listid=insertc(listid, 'Sex="M"', -1);
   listid=insertc(listid, 'and weight > 50', -1);
   call putlist(listid, 'WHERE clause=', 0);
   call send(datid, '_set_where_', listid);
   clearlist(listid);

Get the WHERE clause

   call send(datid, '_get_where_', listid);
   call putlist(listid, 'WHERE List=', 0);
   dellist(listid);
   return;
```

Terminate the classes that were instantiated.
TERM:
    call send(datid,’_term_’);
    return;

_sort

Sorts the table by one or more columns using one or more options

Syntax
CALL SEND (object-id, ’_sort’, column(s), </option(s)>;
<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>column(s)</td>
<td>C</td>
<td>specifies one or more columns by which the table is to be sorted. Columns are separated by blanks. The order of the column names reflects the order in which to sort. The column name should be preceded by the keyword DESCENDING when the column is to be sorted in descending order. Ascending order is assumed unless otherwise specified. You may specify as many columns as you like.</td>
</tr>
<tr>
<td>option(s)</td>
<td>C</td>
<td>specifies one or more sort options to use for the sort. The sort option string must have a slash at the beginning of the string. Sort options are separated by blanks. You may specify as many sort option strings as you like. You can use the options in the following list, depending on your operating system: DIAG, EQUALS, FORCE, LEAVE, LIST, MESSAGE, NODUPKEY, NODUPLICATES, NOEQUALS, OUTPUT, REVERSE, SORTSEQ=ASCII</td>
</tr>
</tbody>
</table>

**Details**

If you modify the current row in the table, you must use the _updateRow method before calling the _sort method.

SAS views cannot be sorted in place. To sort views, you must specify an output table.

If you specify the same table as the output table and the input table, then the _sort method treats this as if no output table were specified. It attempts to sort the table in place. To sort the table in place requires that the table be set to edit mode and that it must be able to be opened in member-level locking.

The _sort method uses the sorting program that SAS supports for your operating system.

The _sort method sets SYSRC for error, note, and warning conditions.

When a permanent or temporary WHERE clause is in effect, the _sort method returns an error.
Example

In the following example, the Data Set Model class is being used as a stand-alone model, that is, the model class is not being used with a data form or data table object.

Load the Data Set Model. In this example, the data set model object is being used as a stand-alone object. Therefore, if you are using SCL similar to this with a data form object or data table object, you should omit the LOADCLASS and INSTANCE functions.

INIT:
    datcl=loadclass('sashelp.fsp.datast_m.class');
    datid=instance(datcl);

Set the data set.

    call send(datid,'_set_dataset_','sasuser.class','edit');

Sort by SEX and NAME in descending order with the NODUPLICATES option.

    call send(datid,'_sort_','sex descending name','
              '/noduplicates');
    return;

Terminate the classes that were instantiated.

TERM:
    call send(datid,'_term_');
    return;

_unlockRow

Unlocks the current row

Syntax

CALL SEND (object-id, '_unlockRow');
Details

If you modify the current row in the table, you must use the _updateRow method before calling the _unlockRow method.

The _unlockRow method uses the last Data Set Data Vector. You cannot call the _unlockRow method until you pass a Data Set Data Vector instance to another method of this object, such as the _getRow method.

The _unlockRow method sets SYSRC for error, note, and warning conditions.

Once the _unlockRow method executes, there is no longer a current row. This is useful in edit mode because a read record is locked.

_updateRow

Updates the current row in the table

Syntax

CALL SEND (object-id, '_updateRow');

Details

To use the _updateRow method, you must be in edit mode. The _updateRow method writes the current row values (passed to the _setRow method) to the data set. You should call the _updateRow method after a call to the _setRow method.

When you move off the current row, the _updateRow method is not automatically called to update the table. The following methods in this class change the current row:

- _addRow
- _copyRow
- _fetchRow
- _findRow
- _getRow
- _gotoAbsoluteRow
- _repeatFindRow
- _setDataset
- _setKey
- _setWhere
- _sort
- _unlockRow.

Therefore, if you modify the current row, you must use the _updateRow method to save the modifications to the table before you call one of the methods listed above. The _updateRow method sets SYSRC for error, note, and warning conditions.
Chapter 61