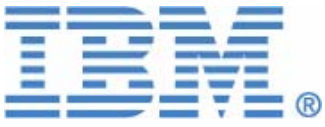
Rational. Statemate

Documentor Reference Guide

Rational Statemate
Documentor Reference Guide



Before using the information in this manual, be sure to read the “Notices” section of the Help or the PDF file available from **Help > List of Books**.

This edition applies to IBM® Rational® Statemate® 4.6 and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Overview of Documentor

You can use the Documentor to design and produce documentation for the system you are designing. With this tool, you can:

- Design your document format.
- Determine what kinds of information to include in it.

Your documents can include textual and graphical information from a variety of sources, including your project database and external files and programs outside your workarea.

You can use the Documentor in conjunction with the Reports tool. Whereas the Reports tool enables you to produce predefined reports, the Documentor enables you to produce reports customized for your specific needs.

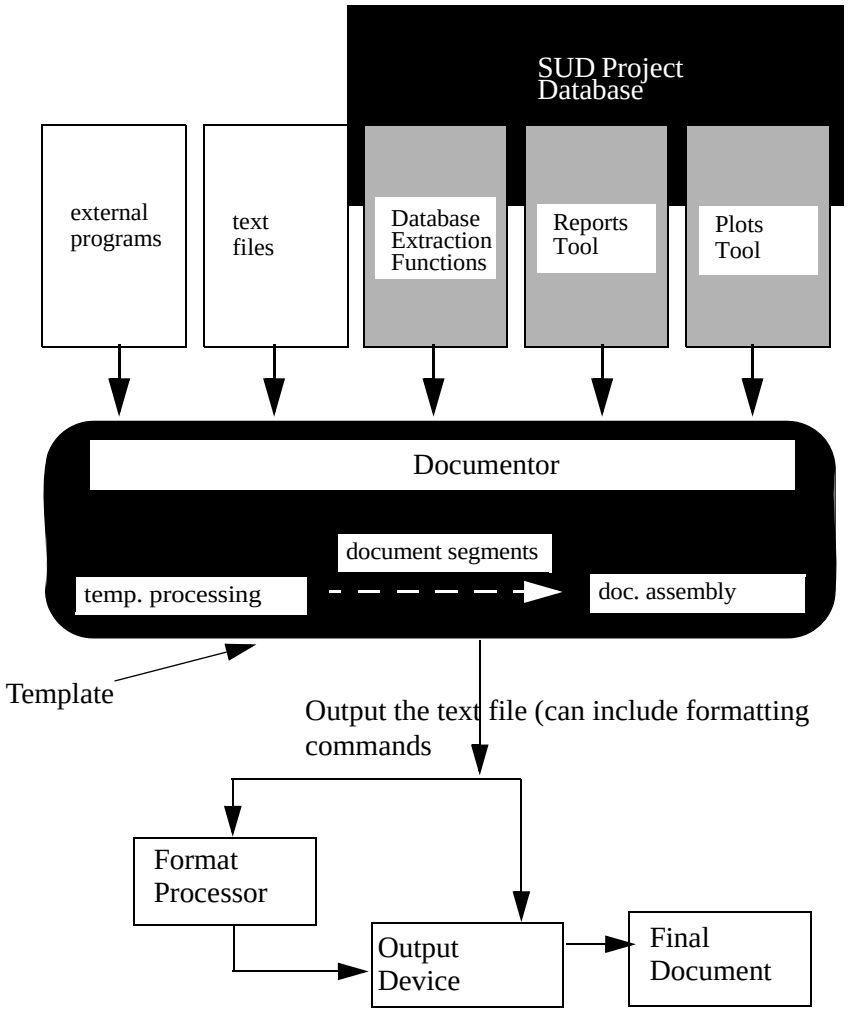
This section presents an overview of the Documentor tool. The topics are as follows:

Basic Concepts

You can use the Documentor to design formatted reports that combine information from the following sources:

- System-under-design (SUD) database retrievals
- Reports produced by the Reports tool
- Plots of charts produced by the Plots tool
- External files (referred to as *include files*) that can contain text, tables, figures, and so on
- Information produced by external programs

The following figure illustrates internal and external sources, and the process of final document production. Note that non-shaded areas represent facilities *outside* of the system.



Document Generation Language (DGL)

You design your documents by writing a program using the Document Generation Language (DGL). This language provides you with great flexibility in designing your document. Among other features, DGL enables you to extract information from your project database (workarea).

DGL Template

The document design program that you write using DGL is called a *template*. The template contains instructions as to what information is to be included in the document. It can also include formatting instructions to be passed to a document format processing system, called a *formatter*. Formatting instructions specify components within your document, including:

- Text width
- Margins
- Headers and footers
- Pagination

DGL Segments

The template is divided into sections called *segments*. To generate the document, you execute the template. Each segment of the template produces a separate text file called a *document segment*.

Creating a document in segments is more efficient than producing an entire document. By using segments, you can produce the document in stages, updating or editing the segments as needed.

Note

The division of the template into segments does not necessarily have to correspond to the final document divisions (chapters, sections, and so on). However, it is a good idea to have the template segments correspond with document divisions wherever possible.

Document Assembly

The Documentor assembles the generated document segments, then does one of the following:

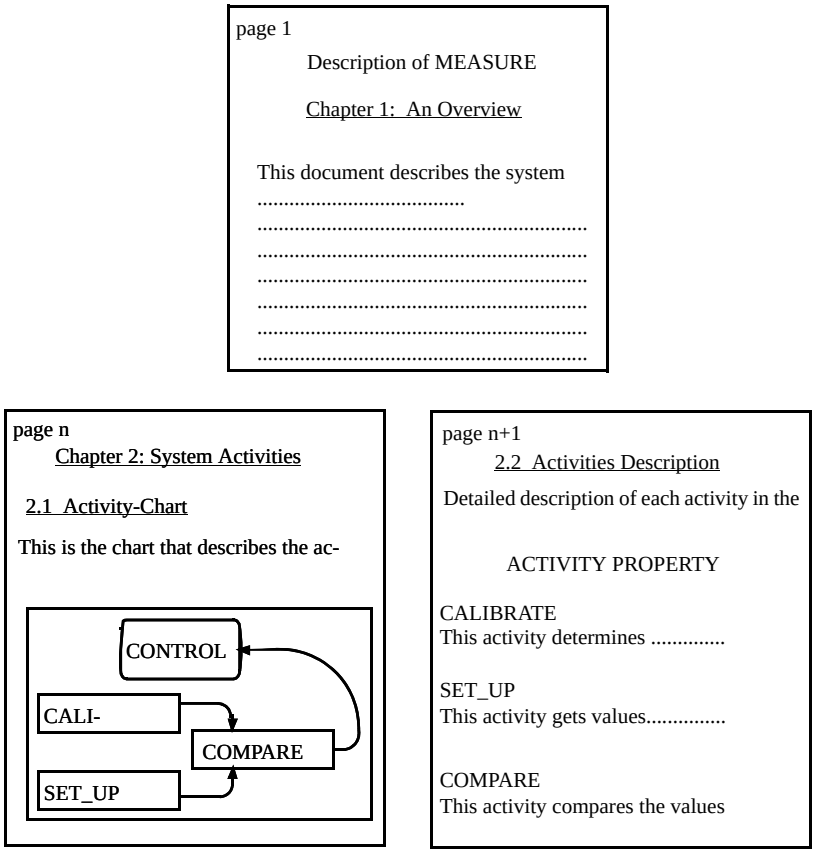
- Exports the segments to an external output device or file.
- Sends the segments to a formatting system to produce the final document.

Designing a Document Using Templates

The following example shows a document that consists of the following:

- Title
- Overview, which contains introductory information from an external text file
- Plot of an activity-chart
- Property information on subactivities shown in the chart, consisting of a property report from the Reports tool

The following figure shows the formatted report.



Generating the Document

The first step is to create the template needed to generate the desired document using DGL. Because the document contains information about a particular activity, you can use a DGL parameter called *act_name*. You specify the value of the parameter (in this case, the activity name for which you are producing the report) when you are ready to generate the document.

Formatting Commands

Some of the commands in the template consist of formatting instructions to be interpreted by the formatter used to produce the final document. This example uses an abstract formatter, rather than a specific one.

The following table lists the commands used by this example.

Command	Description
.skip<n>	Outputs *n* blank lines
.center<title>	Centers the specified title on the line
.chapter<title>	Starts a new chapter with the specified title
.section<title>	Starts a new section with the specified title
.page	Starts a new page
.literal and .end literal	Outputs the block of text specified between the keywords **.literal** and **.end literal**

Sample Template

In a template, statements beginning with "- -" are comments and are not interpreted by the tool. Formatting instructions and other text written between the `/@` and `@/` markers are passed to the output files *verbatim*.

The example uses the following template:

```
TEMPLATE example;

-- Initiation (global) section
PARAMETER
  STRING  act_name;  -- activity for which
                     -- the report is written
VARIABLE
  ACTIVITY act_id;   -- id of "act_name"
  INTEGER st;        -- status return code

BEGIN
  act_id := stm_r_ac (act_name, st);
END;

SEGMENT seg1;    -- contains chapter 1 of the document
```

```
BEGIN
  /@
  .page
  @/
  WRITE  ('.center Description of ',act_name);
  /@
  .chapter An Overview
  @/
  INCLUDE ('sys_overview'); -- an include file containing
                            -- text with formatting commands
END;


SEGMENT seg2;   -- contains chapter 2 of the document

VARIABLE
  LIST OF ACTIVITY  ac_list;

BEGIN
  /@
  .page
  .chapter System Activities
  .section Activity-chart
  .skip
  This is the chart that describes the activities
  of the system:
  @/
  ...........
  stm_plt(act_id, ....); -- Activity-chart plot
  ............
  /@
  .page
  .section Activities' description
  .skip
  Detailed description of each activity in the chart:
  @/
  ac_list:=stm_r_ac_logical_sub_of_ac({act_id},st);
  stm_rpt_dictionary(ac_list,....); -- property report
END;
```

Template Sections

The example template is divided into three parts:

- **Initiation section**—Extracts the activity ID number (*act_id)* from the database. This number is used later in the template to call the appropriate activity-chart plot and to extract the subactivities for the property report.
- **SEGMENT seg1**—Contains instructions to produce the title of the document and chapter 1. In this case, chapter 1 is text contained in an include file, `sys_overview`.
- **SEGMENT seg2**—Contains instructions for producing a plot of an activity-chart and a property report of subactivities.

Executing the Template

After the template is created, you execute it. As part of this process, you enter the value for the parameter *act_name* in a special form provided by the tool. (In this example, the value for the parameter is the activity's name, `MEASURE`.) As a result of specifying the activity name, the appropriate information for the activity `MEASURE` will be included in the document.

The Documentor produces separate text files, corresponding to the segment sections specified in the template. These files include:

- Information generated from the project database (the activity-chart plot and property report).
- Text from an include file.
- Formatting instructions.

Formatting instructions and other text written between the `/@` and `@/` markers are passed to the output files verbatim.

If you have made any errors in your template, error messages are displayed during the execution phase. Correct the template and then re-execute it.

Output Files

After executing the template for the activity `MEASURE`, the resulting output files (document segments) resemble the following:

```
DOCUMENT SEGMENT SEG1:

.page
.center Description of MEASURE
.chapter An Overview
 text and formatting commands as written in the file 'sys_overview'


DOCUMENT SEGMENT SEG2:

.page
.chapter System Activities
.section Activity-chart
.skip
This is the chart that describes the activities of the system:
.literal
 plot information to be sent to the printer
.end literal
.page
.section Activities' description
.skip
Detailed description of each activity in the chart:
Output of Report Tool for the Property Report, including formatting commands for
the specific formatter.
 .
 .
 .
```

Final Assembly

Once the segments are generated, you can use various Documentor options to edit or regenerate them.

When you are satisfied with the segments, you can copy them to an external file or format them. Both of these operations automatically assemble the segments into one file.

The **Format** option assembles the segments and passes them to the format processor that you specify when you create the template. The format processor interprets the formatting statements in the file and produces a final document that is completely formatted.

Reusing Templates

You can generate different documents from the same template using parameters within the template. By changing the values of these parameters, you can change the information that is written, while maintaining the overall structure and format of the document.

For example, the template produces a plot of an activity-chart and a property report for the activity `MEASURE`. Instead of writing the name `MEASURE` directly into the template, the template uses the parameter `act_name`. This way, you can specify the name of the activity in a form at the time of template execution.

You can use the template repeatedly, specifying a different activity name each time you execute the template. This enables you to produce uniform reports for as many activities as desired.

Include Files

You can include files outside of your project database within your documents. In the template example, the introductory `sys_overview` section is not part of the database, but is an external file. You can create such files while you work with the Documentor, or you can copy them from outside your workarea. In the example, this file is assumed to already be in the workarea. Therefore, the file path is not specified—only the file name. The file is accessed via the `INCLUDE` statement within the template and inserted in the segment files when the template is executed.

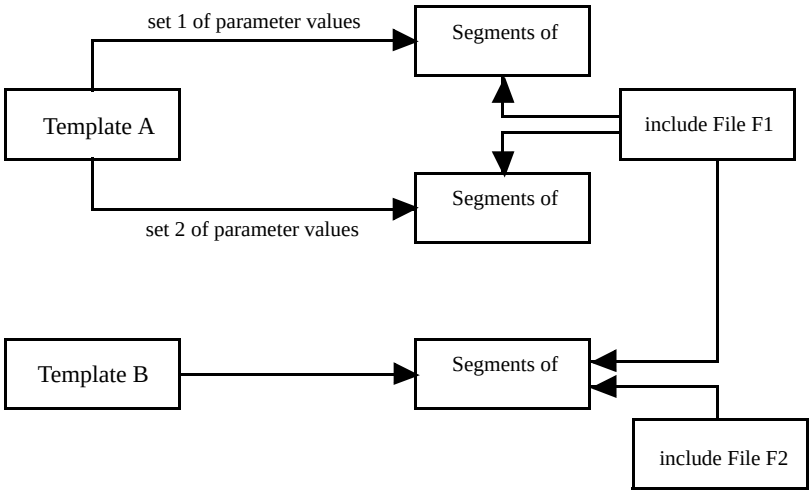
The same include files can be used in a number of different documents, and can be called from a number of different templates.

The figure illustrates the relationship between templates, include files, and parameters, and the segments that are produced after the templates are executed. Template `A` contains parameters and is generated twice—once using set 1 of the parameter values and once using set 2.

Using the first set of parameter values, the segments of document `A1` are generated. In addition, template `A` contains a statement calling an external include file, `F1`, into the generated document segments of A1.

When the second set of parameters is used, the segments of document `A2` are generated. Again, the include file `F1` is included in the output segments.

Template `B` is used to generate the segments of document `B1`. These segments contain two include files: `F1`, which was also included in the segments of document `A1`, and a separate include file, `F2`.



File Access

The Documentor files, templates, and include files used for document generation are stored in your workarea. You create and edit these files in your workarea using the Documentor.

You can store your templates and include files in the databank so they can be shared with other project members, and for version management purposes. In addition, you can export these files from, and import these files to, outside of the system using standard storage functions.

There are templates and include files used to generate standardized documents within a company (or even within an industry) whose use is independent of any particular project. These common files must first be imported to your workarea either directly or via the databank.

Standardized templates, such as templates for the DOD-STD-2167A document set, can be supplied with Rational Statemate. You can import the templates to the project databank when the project is created, or select **File > Import** in the workarea browser.

The resulting segments and documents that the tool creates are not stored in the databank, but reside in your workarea. These are handled the same way as other stored files on your system.

Documentor Interface with Formatting Systems

When you create a template, you can assign a formatter to it from a predefined list (for example, Interleaf).

Embedding Formatting Instructions

You can embed formatting instructions in a template. Upon execution, these instructions are passed verbatim to the output segments. When these segments are assembled and sent to a formatting processor, the instructions are interpreted and a final, formatted document is produced.

You can embed formatting instructions for any kind of formatting system or word processor. Some systems are interactive in nature; that is, you do not directly see the formatting instructions used by the system when working with it.

If you are working with such a system, you must first determine the particular language (set of instructions) that can be used by the system in *batch* mode. These instructions can then be embedded in the template in the same way as for any other formatting system.

Predefined Reports

In many documents, you might want to include a report generated from the Reports tool. The formatting of reports and plots depends on the formatter used and whether this formatter is supported by the system.

Reports from the Reports tool are textual. They have predefined formats determined by embedded formatting instructions. For example, assume that you write a template that calls a report (such as the property report in the sample template), and that you are working with a supported formatter. When you execute the template, the Reports tool automatically embeds the formatting instructions appropriate for the attached formatter within the generated report. This means that the report that appears in the resulting output segments contains embedded formatting commands. Passing the files to the designated formatter results in a document displaying the report in its predefined format. For more information on reports, see **INCLUDE**.

Plots

The Plots tool can generate graphical instructions in several languages. These instructions can be addressed to a printing (plotting) device or to a specific formatter that supports graphics.

For example, the plot in the sample document was included in the document by using a function call `stm_plt`. One of the parameters of this function is the graphical output language. You can pass the attached formatter's language as a parameter to a plot function as long as the formatter can handle a graphical language. In this case, the formatting system processes the plot as part of the entire document.

If you are working with a formatter that cannot process graphical information, you can generate the plot in the language of the output device (printer or plotter) and instruct the formatter to pass the information without processing it. Alternatively, you can produce a separate plot file and later merge the plot into your final document. For more information on plots, see **INCLUDE**.

Note

Plots created using the Word format in the Output Device dialog box are RTF files.

Invoking a Formatter from Within the Documentor

There are two ways to send segment files to a formatter for final processing:

- Export the files to an external file (the export operation automatically assembles the segments into one file). You then invoke the formatter on the file that you want to format as you would for any other text file.
- For several formatting systems, you can choose the **Format** option. This option automatically assembles the segments and sends them to the formatter without requiring you to exit from the Documentor.

Note

The **Format** option is available only for specific formatters, such as nroff. Interleaf, which is an interactive formatting system, cannot be activated from within the Documentor tool.

Using Documentor

This section describes how to use Documentor in detail. It includes:

- Descriptions of Documentor menus and dialog boxes
- Step-by-step procedures for each Documentor option

The first part of the chapter provides an overview of operations, including how to start the Documentor and the connection between the stages of document production and the different tool options. Use this section to locate the menus that you need to perform a particular operation.

The second part of the chapter explains how to perform Documentor operations.

Document Production Process

There are four stages of document production:

Document Production	Menu Option
Writing the template Creating new templates and editing, deleting, copying, exporting, compiling, and printing existing templates.	**Edit > Templates**
Preparing include files Creating new include files and editing, deleting, copying, exporting, and printing existing include files.	**Edit > Include File**
Producing document segments Generating new document segments. The Documentor enables you to regenerate particular segments without having to regenerate the entire document. If you are not satisfied with a segment, you can edit it before producing the final document.	**Edit > Documents**
Producing the final document Regenerating and editing document segments. Deleting, exporting, printing, and formatting existing documents. If the segments do not contain formatting instructions for a particular formatter, you can print the files directly. If the segments contain formatting instructions, you send them to a formatter. This involves either sending the segment files directly to a formatter, or copying them to an external file and formatting them outside of the system.	**Edit > Documents**

Starting Documentor

To start the Documentor from within the system:

1. Start a session and open a project.
2. Click the **Documentor**  icon in the Rational Statemate main window. The Document Management window opens.

   In addition, the Document Management dialog box opens simultaneously so you can easily manage your documents (see **Creating and Manipulating Documents**).

Producing the Document Template

A document template consists of statements (instructions) written in DGL. A template is a text file containing instructions for document generation. In principle, you can create a template in the same way that you create a source file for any other programming language. For simplicity, template files are handled from within the system by standardized storage functions.

Creating and Manipulating Templates

To work with templates, select the **Template**  icon in the toolbar or **Edit > Template** from the main menu. The Template Management dialog box opens. In the dialog box, the **Templates** table lists the available templates in your workarea.

Creating a Template

To create a template:

1. Click **New**. The **New Template** dialog box opens.
2. Enter a name for the new template in the **Name** field. The template name must begin with a letter, and can contain only alphanumeric characters or underscores.

   The name must be unique in this project. If you select different formatters, you can use the same name for multiple templates because the system appends the formatter type to the name you enter, thus making the names unique.

   For example, if you enter the name PAGER and select FrameMaker as the formatter, the system names the template PAGER_FRM. The following table lists the extension for each formatter.

If a template with this name already exists, the Documentor displays a warning. You can cancel or confirm your choice to overwrite the old template. To modify an existing name, click on the cascade button and select a name from the list of templates. This places the name in the **Name** field so you can then modify it.

FrameMaker	_FRM
Glyph	_GLP
Interleaf	_IFF
troff	_TFF
Word	_RTF
Undefined	_OTH

3. Select a formatter from the list of supported format processors.

   The format processor operates on the formatting instructions entered into the template. (See **Overview of Documentor** for more information.) When you copy an existing template, the formatter type for the new template is the same as for the original, unless you specify otherwise.

4. Click **OK**. The text editor opens.

5. Write your template using DGL statements.

6. To save your template, select **File > Save**, then **File > Exit**. to close the text editor. You return to the Template Management dialog box.

You can also create a new template by copying an existing template and editing it as needed.

Editing a Template

To edit an existing template:

1. Select the template from the list in the Template Management dialog box.

2. Click **Edit**. The template opens in the system's text editor.

3. Make your changes, then select **File > Save** and **File > Exit** in the text editor.

Deleting a Template

To delete a template:

1. Select the template from the list in the Template Management dialog box.

   To delete multiple templates, hold down the CTRL key when making your selections.

2. Click **Delete**.

3. The Documentor prompts you to confirm the deletion. Answer **Yes**.

The tool first checks for documents that were generated using the template. If it finds any, the template is not deleted and a list of documents that were generated from the template is displayed in the main window. The Documentor tool assumes that you need to keep the template sources from which existing documents were made. You must delete those documents before their associated templates can be deleted.

Copying a Template

To copy a template:

1. Select the template from the list in the Template Management dialog box.
2. Click **Copy**. The Copy Template dialog box opens with the name of the template in the title bar.
3. Type a new name for the template copy.
4. Select **OK**. A copy of the template is displayed in the Template Management dialog box.

Exporting a Template

To export a template:

1. Select the template from the list in the Template Management dialog box.
2. Click **Export**. The Export Files dialog box opens.
3. Click **OK** to export the template and dismiss the dialog box.

Compiling a Template

You can check a template for adherence to DGL syntax rules using the **Compile** option. This is the same as program compilation in other languages. Use this option to see whether a template is executable, without actually performing the execution. Any errors are logged so you can execute templates unattended.

Note

If you do not compile your template, the tool checks the template for errors upon execution (see **Creating a Document**).

To compile a template:

1. Select the template from the list in the Template Management dialog box.
2. Click **Compile**. A message is displayed in the main Documentor window that notifies you whether the compilation was successful.

If there are compilation errors, a message notifies you that errors were found. The error messages are written into the template in the form of comments. A compiler error message appears as close as possible to the line where the problem was found. The following example shows a template with error messages:

```
    .
    .
    .
name := stm_r_st_name (state_chart, status;
--%DOC (E1352) Missing ')'
name := name + 5;
--%DOC (E1142) Argument mismatch
name := nm;
--%DOC (E1211) Identifier NM not declared
    .
    .
    .
FOR sub_state IN sub_list LOOP
i:=i+1
--%DOC (E2001) Missing ';'
    .
    .
    .
```

You must correct template errors before re-executing the template, but you do not need to remove them from the template file. The Documentor automatically removes them the next time the template is compiled or executed.

Printing a Template

To print a template:

1. Select the template from the list in the Template Management dialog box.
2. Click **Print**.

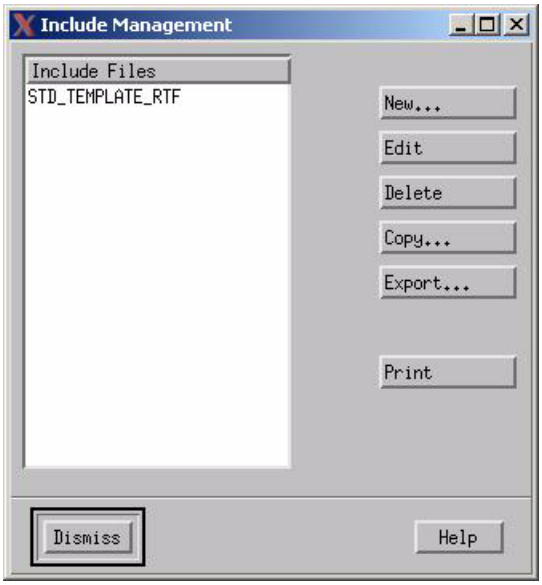
To print multiple templates, hold down the CTRL key when making your selections.

Using Include Files

Your document might include files external to the specification database. Such include files consist of textual explanations, diagrams, pictures, and so on. Insert these files in your document using the INCLUDE statement in the template. When you execute the template, the include files are copied into the resulting document segments.

Creating and Manipulating Include Files

To add include files, click the **Include Files**  icon in the main Documentor window, or select **Edit** > **Include File**. The Include Management dialog box opens, as shown in the following figure.

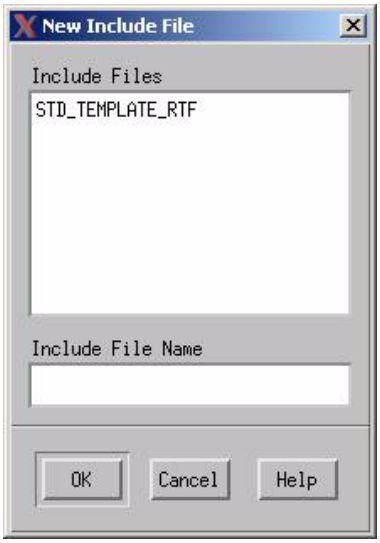


The dialog box lists the include files in your workarea.

Creating an Include File

To create an include file:

1. Click **New**. The **New Include File** dialog opens.



2. Enter a name for the new include file in the text box. The include file name must begin with a letter, and can contain only alphanumeric characters or underscores.

   The name must be unique in this project. If an include file with this name already exists, the system displays a warning. You can cancel or confirm your choice to overwrite the old include file. To modify an existing name, select a name from the list and modify it in the text box.

3. Click **OK** to invoke the system's text editor where you can create a new include file.

Alternatively, you can create a new include file by copying an existing include file and editing it as needed.

Editing an Include File

To edit an include file:

1. Select the include file from the list in the Include Management dialog box.
2. Click **Edit**. The include file is displayed in the system's text editor.
3. Make your changes, then select **File > Save** and **File > Exit** in the text editor.

Deleting an Include File

To delete an include file:

1. Select the include file from the list in the Include Management dialog box.

   To delete multiple include files, hold down the **CTRL** key when making your selections.

2. Click **Delete**.
3. The Documentor prompts you to confirm the deletion. Answer **Yes**.

Copying an Include File

To copy an include file:

1. Select the include file from the list in the Include Management dialog box.
2. Click **Copy**. The Copy Include dialog opens with the name of the include file in the title bar.
3. Type a new name for the include file copy.
4. Click **OK**. A copy of the include file is displayed in the Include Management dialog box.

Exporting an Include File

To export an include file:

1. Select the include file from the list in the Include Management dialog box.
2. Click **Export**. The Export Files dialog box opens.
3. Type the directory and file name where you want to export the include file.
4. Click **OK** to apply your changes and dismiss the dialog box.

Printing an Include File

To print an include file:

1. Select the include file from the list in the Include Management dialog box.
2. Click **Print**.

To print multiple include files, hold down the `CTRL` key when making your selections.

Producing the Document Segments

After writing the template and preparing any include files to be included in the document, you are ready to execute the template and generate the *unformatted* document segments. Each segment file contains information from various sources that you specified through the DGL statements in your template.

The Documentor permits file operations on segments independent of the sources from which they were generated. These operations work in the same way for both for templates and include files.

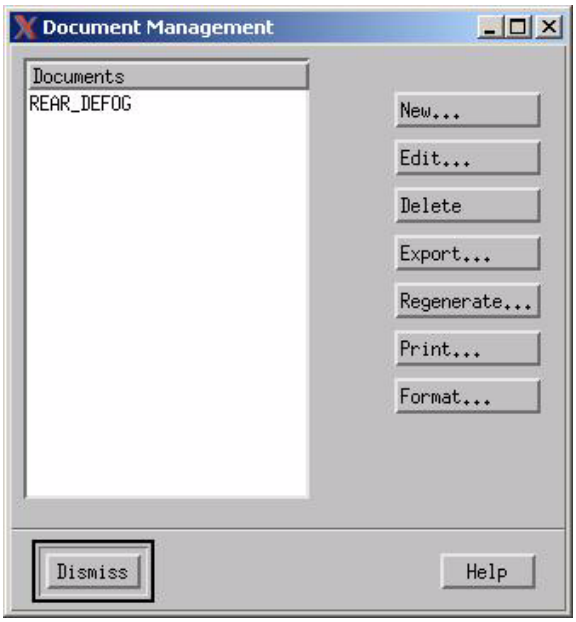
This section describes how to use the following options in the Document Management dialog box:

- **New**—Generates segments for a new document
- **Edit**—Modifies any of the generated segments
- **Delete**—Deletes a document
- **Regenerate**—Regenerates specific segments of the document without re-executing the entire document.

For information on the **Export**, **Print**, and **Format** options, see **Producing the Final Document.**

Creating and Manipulating Documents

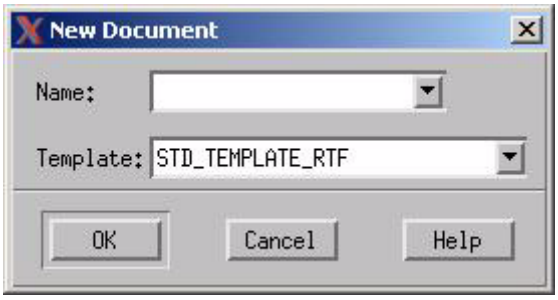
To work with documents, click the **Documents**  icon in the toolbar or select **Edit > Documents** from the main menu. The Document Management dialog box lists the documents in your workarea.



Creating a Document

To create a document:

1. Click **New**. The **New Document** dialog box opens.



2. Enter a name for the new document in the text box. The name must begin with a letter, and can contain only alphanumeric characters or underscores.

   The name must be unique in this project. If a document with this name already exists, the system displays a warning. You can cancel or confirm your choice to overwrite the old document. To modify an existing name, select a name from the drop-down list and modify it in the text box.

3. Enter the name of the template you want to execute, or select it from the list of templates.

4. Click **OK**.

   The Documentor checks the template for errors. If there are compilation errors, the tool displays a warning. You must correct all the errors before you can generate any segments. See **Creating a Template** for more information. When all of the errors have been resolved, click **New** again.

   If there were no errors, the Generate Document dialog box opens, as shown in the figure. This dialog box enables you to select all or some of the template segments and enter values for the parameters to be used when generating the document.

   The Generate Document dialog box contains two tables:

   a. **Parameter Name**—Displays the current parameter values. If you assigned initial values to the parameters in the template, these values are displayed in the **Parameter Value** column. If you change these values, the new values are stored and displayed prior to any regeneration of the document segments. (For more information, see **Regenerating Document Segments**.)

   b. **Segment Name**—Lists the document segments, all of which will be generated by default (see the Yes value in the **Gen** column). If you do not want to generate a specific segment, click on the Yes next to that segment and the field changes to No.

      By default, the view is collapsed so the segment information is not displayed. Click **Expand** to expand the view.

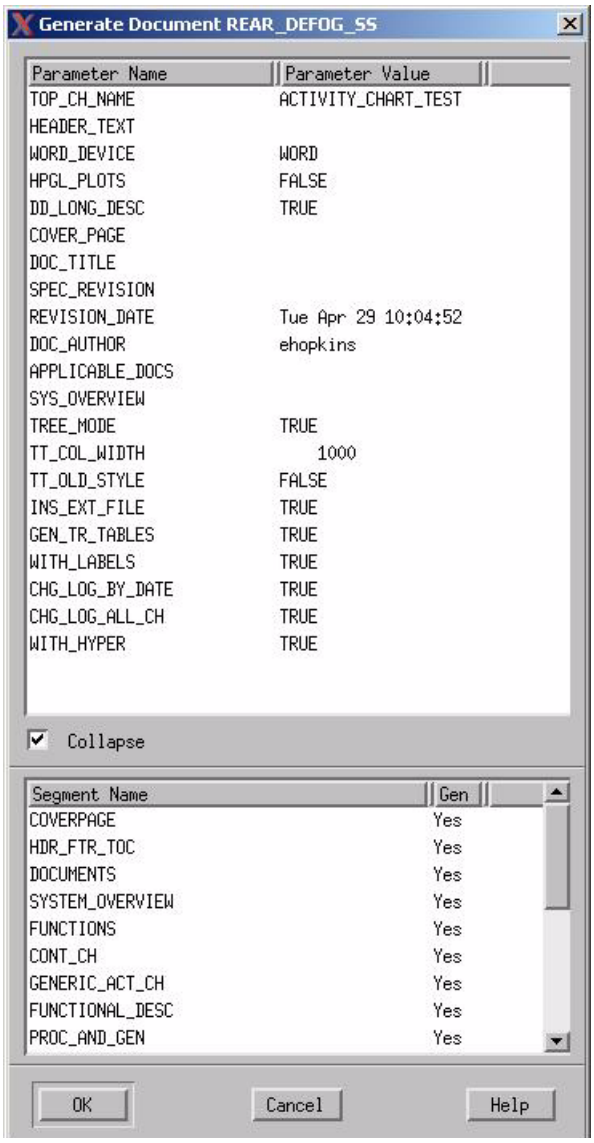
      Regardless of the template segments that you select for execution, the initiation section of the template is always executed. See **Overview of Documentor** for an explanation of template segments.

5. Enter values for the relevant parameters shown in the form. (For more information on template parameters, see **Reusing Templates**.)

6. Click **OK** to apply your changes and dismiss the dialog box.

The tool generates the segments and the new document name is included in the list in the Document Management dialog box.

Generate Document Dialog Box (shown expanded)

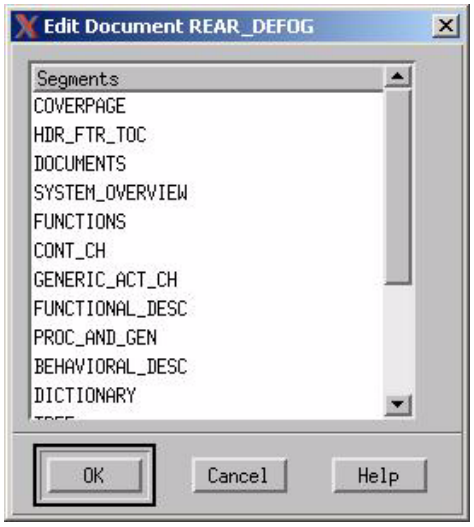


Editing a Document

After generating the document, you can edit any of the generated segments. The **Edit** option enables you to make minor changes in a generated document segment, without having to edit and re-execute the entire document template.

To edit a document segment:

1. Select the document from the list in the Document Management dialog box.
2. Click **Edit**. The **Edit Document** dialog box opens, as shown in the following figure.



3. Select the segment you want to edit.
4. Click **OK**. The segment opens in the system's text editor.
5. Make your changes, then select **File > Save** and **File > Exit** in the text editor.

Deleting a Document

To delete a document:

1. Select the document from the list in the Document Management dialog box.

   To delete multiple documents, hold down the CTRL key when making your selections.

2. Click **Delete**.
3. The Documentor prompts you to confirm the deletion. Answer **Yes**.

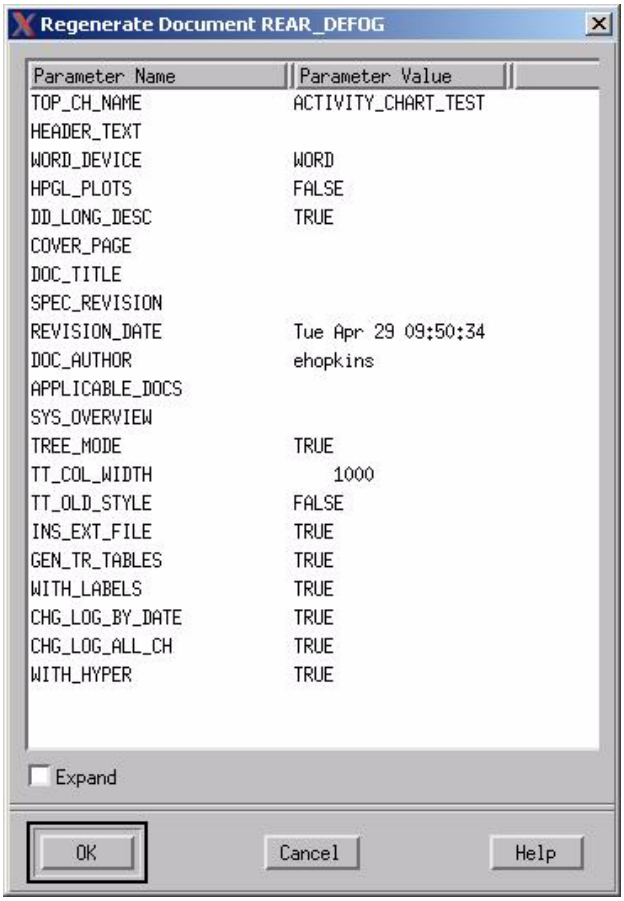
Regenerating Document Segments

The Documentor enables you to regenerate particular segments of your document without re-executing the entire template. After generating some of the document's segments, you might want to go back and generate other segments that you did not originally select. Perhaps you altered the template or changed your specification. You will need to regenerate those segments affected by the modifications.

Using the **New** option for this task is inefficient. When you create a new document, any previous segments for the document are erased, then the specified segments are generated. This is why Documentor provides an additional document operation, **Regenerate**. This option allows you to redo particular segments, without affecting existing segments of your document.

To regenerate a document segment:

1. Select the document from the list in the Document Management dialog box.
2. Click **Regenerate**. The Regenerate Document dialog box opens.



3. By default, all segments are selected. Unselect the segments that you do *not* want to regenerate by clicking on the Yes next to it; the field changes to No.

You can change parameter values. However, changing the parameter values can be problematic when regenerating segments. It is usually undesirable to have some segments generated with one set of parameter values, and other segments created with a different parameter set. If you alter some parameters and regenerate a portion of your document, the Documentor asks you to confirm the regeneration.

4. Click **OK** to apply your changes and dismiss the dialog box.

Producing the Final Document

After generating the document segments, you output them as an assembled finished document. If the document segments contain formatting instructions, you send them to a formatter to produce the final document.This procedure also depends on whether the Documentor can invoke the formatter. For example, Interleaf cannot be activated from within the Documentor.

Select one of the following three ways to produce the final document depending on your formatter:

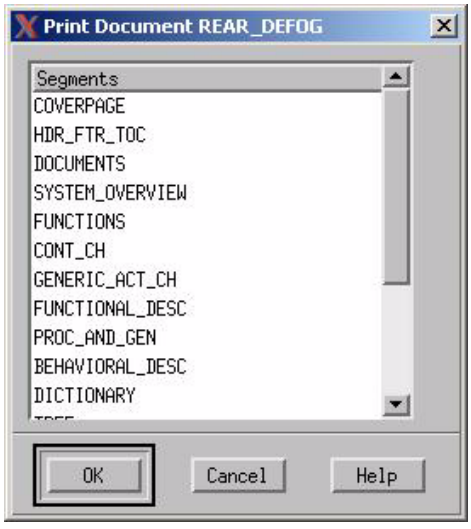
- For ASCII and PostScript files, use **Print**.
- For FrameMaker, Interleaf, Word, and troff files, use **Export**.
- For Glyph files, use **Format**.

Printing a Document

Use **Print** for document segments that do *not* contain formatting instructions (such as ASCII and PostScript) that you want to send directly to your local printer. This option is also useful for shipping a document in a machine-readable format to another site.

To print a document:

1. Select the document from the list in the Document Management dialog box.
2. Select **Print**. The **Print Document** dialog box opens, as shown in the following figure.



3. Select the segments you want to print:
   - Select one segment by clicking on it.
   - Select multiple segments by holding the CTRL key.
   - Do not select any segments to print all the segments.
4. Click **OK** to dismiss the dialog box and print the document.

Exporting a Document

Use **Export** for document segments that contain formatting instructions for formatters that cannot be activated from within the Documentor (such as FrameMaker, Interleaf, Word, and troff).

To export a document to an external file:

1. Select the document you want to export from the list in the Document Management dialog box.
2. Click **Export**. The Export Document dialog box opens with the name of the document you selected in the title bar. It lists all the generated segments for the selected document.
3. Select the segments you want to export:
   - Select one segment by clicking on it.
   - Select multiple segments by holding the CTRL key.
   - Do not select any segments to export all the segments.
4. Click **OK** to dismiss the dialog box.
5. Enter the output file path name (using the operating system conventions for your host computer). This is the file destination for the output.

   Note: You can also enter a new name by clicking on the ellipsis (...) to display the **Export Files** dialog box.
6. Click **OK**. The tool assembles the segments in the correct order and writes them to the specified destination.
7. Invoke your formatter on that external file.

Formatting a Document

Use **Format** for document segments that contain formatting instructions for formatters that can be activated directly from within the Documentor (such as Glyph).

To format a document:

1. Select the document from the list in the Document Management dialog box.
2. Click **Format**. The Format Document dialog box opens.
3. Specify the field values:
   - Select the segments that you want in your document.
   - Optionally, select additional output devices **Terminal**, **Printer**, or both.
   - Specify the output file path name. This is the file that contains the final document. The file is produced by the formatting system.
4. Click **OK**. The Documentor assembles the segments, then activates the formatter to produce the document.

Working with Different Formatters

Overview of Documentor explains the relationship between the Documentor and formatters. To use any formatter, it must be present on your host computer. You can edit or generate Documentor files on any computer running Documentor, but you can use the **Format** option only on a host that actually runs the target formatting system.

Using nroff and troff

nroff and troff are UNIX-based formatting systems.

nroff is a textual formatting system that prepares output for standard ASCII devices such as terminals, disk files, and printers. You can activate nroff directly from the Documentor using the **Format** option on templates attached to the troff formatter.

troff is a graphical formatting system that prepares output for laser printing devices. Despite the difference, these systems are compatible in that troff can use nroff input files to produce final output. troff cannot be activated directly from the Documentor.

Note

- nroff input requires preprocessing for commands used with equations and tables. If you use such commands in your template or include files, you should *not* activate nroff directly from the Documentor. The Documentor does *not* support preprocessing of output before sending it to nroff.
- If you run nroff externally, predefined reports generated by the Reports tool use the macros library `me`.

For information on troff and nroff, see the UNIX system documentation for your computer.

Using Interleaf

Interleaf is a documentation preparation system available on various computer systems. Because it is interactive, Interleaf cannot be activated directly from the Documentor. In addition, you must make sure to include the file `interleaf_glob` in the first segment of your template. The file is supplied as an include file in the databank. You must load this include file into your workarea for Documentor to access it. This file contains global definitions that are used in Rational Statemate reports.

For more information on Interleaf, see the Interleaf documentation provided with the formatter.

Document Templates

A template is a text file consisting of instructions for generating a document. The instructions are written in the Document Generation Language (DGL). This section, divided into three parts, explains the fundamentals of DGL:

- Principles of DGL
- Data-items and expressions
- Detailed explanation of DGL statements

Part one covers the basic concepts of DGL and shows you the general principles of template writing. In part two, we discuss identifiers and the types of data they can represent. We also detail the various kinds of expressions that you can use in DGL statements. In part three, we present each DGL statement.

In your template, you can call functions that extract the information stored in the specification database, and write this information into your document. The use of Database Extraction Functions is covered in **Overview of the Extraction Functions**.

Principles of DGL

DGL is a structured programming language and displays features typical of other structured languages such as Pascal. Among these are: declarations for identifiers, the use of variables, parameters and constants, and control flow statements. In this section, the principles and conventions of DGL are introduced.

DGL Template Structure

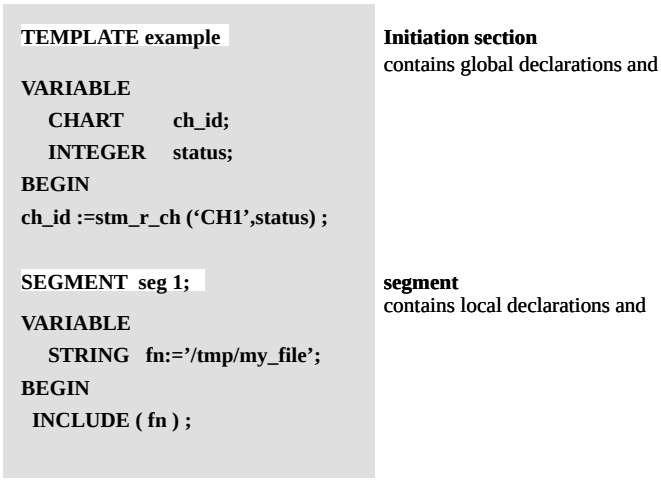
Every template consists of two main parts:

1. Initiation section—Contains declarations and statements that pertain to the overall template. It may not include output statements.
2. Template segments—Sections of the program, each of which pertains to a particular portion of the document.

```
TEMPLATE example

VARIABLE
    CHART       ch_id;
    INTEGER     status;
BEGIN
ch_id :=stm_r_ch (‘CH1’,status) ;

SEGMENT  seg 1;

VARIABLE
    STRING   fn:=’/tmp/my_file’;
BEGIN
 INCLUDE ( fn ) ;
```

Initiation section
contains global declarations and

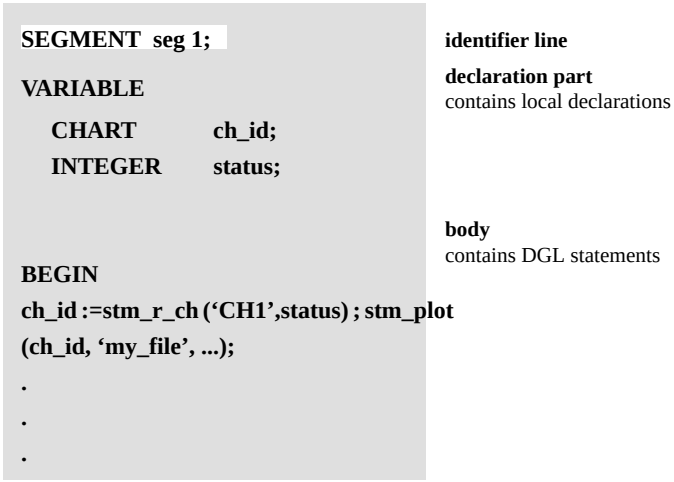
segment
contains local declarations and



Template segments are the basic divisions of the document and, when executed, produce distinct output files called *document segments;* these are later assembled into the final document. The segments do not necessarily correspond to the document divisions (i.e. chapters, sections, etc.). However, we recommend that you divide the segments according to such divisions. This gives you the ability to generate separate sections or chapters individually; the initiation section, however, is executed whenever any segment is executed. See the discussion on **Template Sections** for more information.

The initiation section and template segments all have the same overall structure:

1. **Identifier line** - (Required) Identifies the entire template or template segment by name.
2. **Declaration part** - (Optional) Contains the definitions of any constants, or variables used by the template section during execution. Identifiers declared in the initiation section are global, i.e., they may be used throughout the template; in the initiation section (only) you may include parameter declarations, as well as variables and constants.
3. **Body** - (Required) Contains execution statements. The body is composed of a BEGIN/END statement that delineates the section's statements. Any number of statements may be found in the body. The initiation section may **not** include output statements.



DGL Syntax Rules

DGL, like most programming languages, has a particular syntax that must be obeyed. The full syntax of each DGL statement is given in **DGL Statement Reference**. A more formal DGL syntax, written in BNF, is provided in **BNF Syntax**.

The following is a list of general syntax rules that apply to all DGL statements:

1. DGL is not case sensitive. The exception to this rule is literal strings (inside apostrophe marks). These are utilized exactly as they appear in the template.
2. DGL statements terminate with a semicolon.
3. Multiple statements on a single line are permitted.
4. A single statement may span several lines.

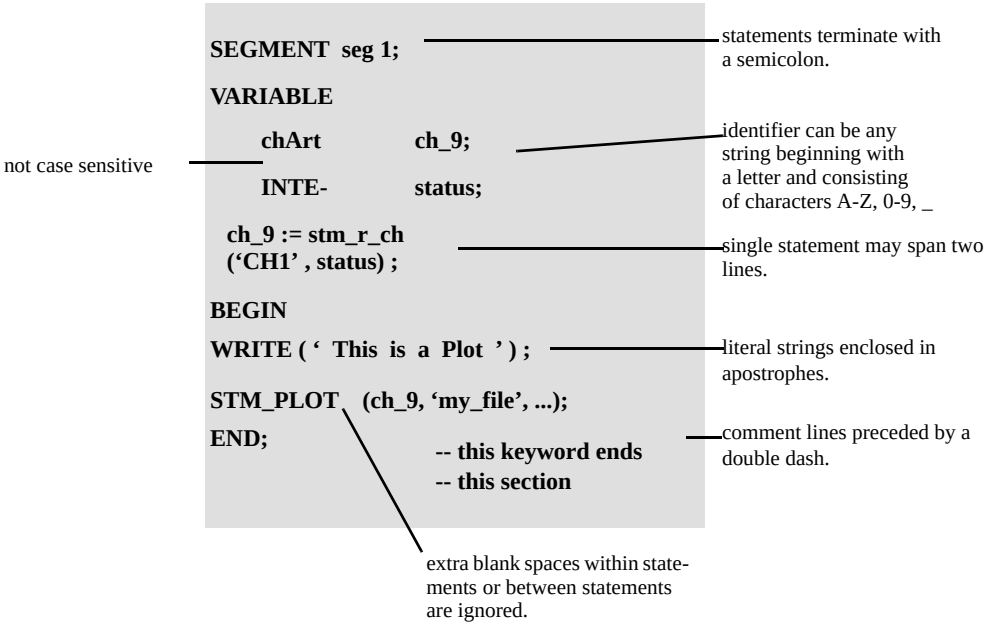
5. DGL has a set of reserved words and syntactical constructs. Each of these has a special meaning and can be used only in the context for which it has been designed. The DGL reserved words are listed in **DGL Reserved Words**.

6. An identifier (a name you assign to an object for identification purposes) can be any string, beginning with a letter and consisting of any of the following characters: A-Z, a-z, 0-9, _. Reserved words may not be used as identifiers.

7. Literal strings may be assigned by enclosing them within apostrophe marks ('). Any character sequence can be used inside literal strings.

8. Comments are preceded by a double dash (--) symbol. This symbol can be placed anywhere in the line except within a literal string. After the comment symbol, all other characters on the line are ignored.

9. Extra blank spaces within or between statements are ignored.



Special Features of DGL

We mentioned before that DGL resembles other structured programming languages. DGL also includes special constructs and features important for the document generation process.

Extensions to Conventional Programming Constructs

Some of the features unique to DGL are:

- **Verbatim Inclusion**—Text may be included in a template and passed “as is”, i.e. literally to the output document segments. This is particularly useful for passing formatting commands to the formatter.
- **Include File**—Text from another file may be copied to the output document segments.
- **Calling External Programs**—External programs (e.g., operating system services) may be called from within the template.
- **Include Statemate Reports and Plots**—Rational Statemate predefined reports and graphical plots may be included in the document.

Database Extractions

A set of functions can be used to extract database information regarding specific elements and produce lists of elements according to specified criteria. Database extraction functions and their use are explained in **Documentor Functions**.

There are several kinds of functions:

- **Single-Element Functions**—Return element details (as strings, or numbers). Such functions can be used to return an element name, synonym, type, definition, short description, or attribute values. Parts of an element’s long description may be retrieved through the use of keywords.
- **Query Functions**—Retrieve a list of elements having a certain relationship with other specified elements or having a specified attribute value. For example, you may retrieve all descendants of a given activity and store them in a list.
- **Utility Functions**—Perform operations and manipulations (e.g. sort) on lists, single integers and strings.

Overview of DGL Statements

Here is a brief overview of DGL statements. Subsequent sections provide you with the full range of available statements and their syntax. DGL is written as statements, each of which is a specific command to the Documentor Tool. The types of statements vary, and most are similar to constructs in other languages.

DGL statements consist of the following types:

- **Structure statements**—Define the structure of the template.
- **Declaration statements**—Declare the type of identifier for variables, constants and parameters.
- **Assignment statement**—Assign a value to a variable.
- **File handling statements**—Open or close files or the dialog area, and read data from files.
- **Output statements**—Pass information to output document segments, or to the dialog area.
- **Control flow statements**—For conditional and iterating execution of statements.

```
SEGMENT  seg 1;                                  structure statement
VARIABLE                                         declaration statement
    CHART          ch_id;
    INTEGER        status;
BEGIN                                            structure statement

 ch_id :=          stm_r_ch ('CH1' ,status) ;    database extraction function in
                                                 an assignment statement
 IF status = stm_success                         control-flow statement
  THEN                                           output  statement
   WRITE ('This is a Plot');                     output  statement (plot statement)
   STM_PLOT(ch_id, 'my_file', ...);              control-flow statement
  ELSE                                           output  statement
  WRITE('Retrieval    failed'  );
 END IF;                                         structure statement
END;
```

Data-types and Expressions

Identifiers are names that are used in a template and can represent constants, variables or parameters. The differences between these are noted here:

- **Constants** are identifiers whose values are constant and cannot be changed in DGL statements.
- **Variables** are identifiers whose values can change in DGL statements.
- **Parameters** are variables whose values may be assigned in a special form before executing the template.

Before you use an identifier, you must declare whether it is a constant, variable or parameter. Furthermore, you must declare the particular *type* of value that can be assigned to the identifier; we call this type a *data-type*.

For instance, an identifier (whether constant, variable, or parameter) may be declared to hold *integer* values, *float* values, *string* values, etc.

Identifiers can be combined to construct expressions; these can be used in various kinds of statements and database extraction functions. Various kinds of expressions can be constructed, depending on the data-types of the identifiers of which they are constituted.

For instance, you may construct numeric or string expressions as well as Rational Statemate element expressions.

This section deals with both data-types and the expressions you can construct from them. In addition, *enumerated types*; variables or return values of a function that take a restricted number of discrete values, are described.

Data-types

You declare identifiers and their data-types in declaration statements; these are described in later sections. Below we list the various data types that are recognized in DGL declaration statements.

Conventional Types found in Other Programming Languages

INTEGER (numeric)
FLOAT (numeric)
Boolean
STRING
FILE

Rational Statemate Element Types

These data-types are Rational Statemate elements. Typically, you use identifiers of this type in Rational Statemate database extraction functions. For example:

```
ACTIVITY    ac;
STRING      ac_name;
   .
   .
ac := stm_r_ac (ac_name , st);
WRITE('activity synonym is ',
     stm_r_ac_synonym(ac,st));
```

The variable ac is declared here to be of type ACTIVITY.
A database extraction function assigns a value to ac (the value is an activity whose name is the value of the variable ac_name). In the WRITE statement that follows, ac is used as an argument in another function that returns the synonym for the activity represented by ac.

There are two kinds of type declarations for Rational Statemate elements:

- **ELEMENT** - this declaration allows you to assign any Rational Statemate element to the identifier.
- **Specific Statemate element Types** - The following types are recognized:

```
ACTION
ACTIVITY
A_FLOW_LINE
CHART
CONDITION
CONNECTOR
DATA_ITEM
DATA_STORE
DATA_TYPE
EVENT
FIELD
FUNCTION
INFORMATION_FLOW
M_FLOW_LINE
MODULE
REQUIREMENT
STATE
TRANSITION
```

The variables having Rational Statemate element data-type hold an ID number which is used for internal representation.

You may use identifiers of type ELEMENT in place of using identifiers with specific type declarations. For instance, in the previous example, we could have legitimately declared the variable ac to be of type ELEMENT instead of type ACTIVITY.

However, there are advantages to using specific element type declarations. When you use a specific element type declaration, identifier assignments and function parameters are checked to ensure that values are of the proper type. This is of particular importance for assignments by database extraction functions. For instance, in this above example, the function `stm_r_ac` returns an activity ID that is assigned to `ac`; if we use `ac` with a function call that returns a module, the Documentor detects this as an error during template compilation. If, however, we had declared `ac` to be of type ELEMENT, no syntax error would have been detected.

In some cases, you must declare an identifier to be of type ELEMENT instead of being of a specific element type. This case comprises identifiers that are assigned values from a list of elements of mixed type (i.e., of more than one element type). For an illustration of this, see the example below for type LIST OF ELEMENT.

LIST OF simple_type

Identifiers declared as LIST OF *simple-type* can be assigned values of a list of items of any data-type. Simple-type may be any of the above types (e.g. LIST OF STRING, LIST OF ACTIVITY, etc.).

The following example shows a typical use of identifiers of this type; in this case, we have declared an identifier of type LIST OF STATE.

```
VARIABLE
    LIST OF STATE     sub_st;
    STATE             st_id;
    INTEGER           st;
BEGIN
st_id := stm_r_st (‘S1’ , st) ;
sub_st := stm_r_st_logical_sub_of_st({st_id},st) ;

END;
```

In this example, the variable `st_id` is assigned the ID of the state `S1.`

This variable is then used with another database extraction function to extract all of the substates of `st_id` and assign their values to the variable `sub_st` of type LIST OF STATE.

Expressions

Parameters, variables and constants are used to build expressions, such as `a + 5.`

Expressions such as these may be used in assignment statements, in calls to predefined functions, and in Boolean expressions (comparisons). In addition, functions themselves may participate in expressions.

Expressions can be constructed for any of the data-types:

- **NUMERIC**: for example, `a*(b+3.2)`
- **STRING**: for example, `'DATA-ITEM DICTIONARY'`
- **Boolean**: for example, `A > B`
- **Statemate element**: for example, `stm_r_ac ( 'AA' , st)`
- **LIST**: for example, `a_list + b_list`

In the following sections, we present a more detailed explanation of the expression types that you can construct.

Numeric Expressions

These are integer and real numbers. Their constant values are the same as in conventional programming languages.

Binary operations (+ , - , * , / , **) and unary operations (- , +) follow conventional precedence rules.

Numeric expressions may mix integer or real operands.

Parenthesis may be used to change the precedence of operations.

Boolean Expressions

The basic Boolean expressions are comparisons between expressions of other types. Note that not all comparisons are legal for all data-types.

Operator	Meaning	Allowed types
=	Equality	all
<>	inequality	all
<	Less than	numeric and lists
<=	Less than or equal to	numeric and lists
>	Greater than	numeric and lists
>=	Greater than or equal to	numeric and lists

For LIST types, Boolean comparisons are understood in terms of inclusion. For example, for lists A and B, A < B if B contains all elements of A and also other elements.

The Boolean operations NOT, OR, and AND are also supported.

In addition there are two predefined Boolean constants: TRUE and FALSE.

String Expressions

String literals are written within apostrophes, for example: `'ABC'`

Spaces within literal strings are always considered. For example, `'A  BC '` is different from `'ABC'`. String literals may contain formatting characters using the backslash character (\) character:

- `\n`: inserts a new line in the string.
- `\t`: inserts a tab in the string.
- `\`: the backslash, when followed by any other character, includes that character literally into the string. This is not intended for use with alphanumeric characters but for including special characters in the string - especially \ and `'`.

String expressions may include string constants, string variables, and functions that return strings.

Concatenation of strings is supported; it is indicated by the “+” sign. For example: `'str1' + 'abc'` results in `'str1abc'`.

Rational Statemate Element Expressions

Expressions of this type may be either declared variables or function calls that return a Rational Statemate element (refer tp **Documentor Functions**). For example, `md := stm_r_md (md_name , st);` returns a value of type MODULE and assigns it to the variable `md` in the assignment statement.

As another example, consider the following:

```
stm_r_md_synonym (stm_r_md ('M1' , st1) , st2)
```

Here we use the module extracted by the function `stm_r_md` as an argument in another function.

There are no constants for Rational Statemate elements. The variables of the above types get their values via the Rational Statemate predefined functions, and are used as arguments in other predefined functions to retrieve additional information. Rational Statemate element ID numbers are used as values of these expressions.

List Expressions

This type is used to handle collections of items of any of the DGL types. A list is created either by explicitly enumerating the items in the list, or as a result of a function call. Explicit enumeration is written as:

```
{ list_item , list_item , ... }
```

as in the string list: `{'abc', 'def', 'xyz'} .`

A list item must be an expression of the list type.

All items in the list must be of the same type. Lists can be created from other lists using the operations: *union* (+), *subtraction* (-), *intersection* (*), and *concatenation* (&).

This last operation differs from *union* when two identical lists are used, for instance:

`{'Alpha'}+{'Alpha'}` produces `{'Alpha'}`,
while `{'Alpha'} & {'Alpha'}` produces `{'Alpha','Alpha'} .`

A list expression can also be built as a result of a function. For example,

```
stm_r_md_name_of_md ( 'M*' , st) .
```

The value of this list expression is all modules whose names begin with `M`.

Enumerated Types—Predefined Constants

A variable or return value of a function is considered of *enumerated type* if it may take a restricted number of discrete values. For example, a variable that represents a day of the week may have only seven values, Sunday through Saturday.

DGL does not directly support enumerated types. The way to deal with variables of such a type is to declare them as INTEGER, and to define constants that are equal to the specific possible values. In order to make the template clearer, you may use meaningful names for these constants. For instance, you may define constants for the days of the week: SUNDAY:= 1, MONDAY:= 2, and so on.

The Documentor has several sets of *predefined* constants used as enumerated types. The names for these constants always begin with the prefix *stm_*.

For example, a very useful enumerated type is Element Type. The possible values for this type are *stm_state*, *stm_event*, etc; each of these identifiers has a unique integer value.

You may use predefined constants in your template without even knowing their numerical values. However, you must make sure that variables that are to be assigned enumerated values are declared as INTEGER.

For example, another widely used predefined enumerated type is the function *return status code* (see **Documentor Functions**). Status codes have a restricted number of values, each value denoting some information about the operation of the function being used.

For example, *stm_success*, denoting the successful completion of the function operation, corresponds to the value *0*. You do not have to explicitly use the value *0* when writing this status code into your template; for instance, you can write:

```
VARIABLE
   INTEGER   status ;
        .
        .
   md_id := stm_r_md ('M1' , status);
   IF status = stm_success THEN
        .
        .
```

Predefined enumerated types are listed in their relevant sections.
For instance, an enumerated type that is returned by a certain function is listed in the description of that function.

DGL Statements

This section presents the DGL statements and their syntax, along with examples of their use.

Structure Statements

These statements define the structure of the template. There are four statements that define the structure of your template:

- **TEMPLATE statement**—First statement of the template.
- **SEGMENT statement**—Starts a new segment section.
- **PROCEDURE statement**—Starts a new procedure section.
- **BEGIN/END statement**—Marks the boundaries of a template section.

In addition, Comment statements can be used to indicate program comments.

Next is a more detailed discussion of each structure statement.

TEMPLATE Statement

Statement Syntax:

```
TEMPLATE template_name ;
```

The TEMPLATE statement is the first statement in the template, and assigns an identifying name to the template. This name is used for internal documentation purposes only and does not have to correspond to the name you use to designate the template in the *Create Template* form.

SEGMENT Statement

Statement Syntax:

```
SEGMENT segment_name ;
```

The SEGMENT statement starts a new segment section. It is the first statement of a segment, and assigns an identifying name to it. The identifier is limited to 16 characters maximum. The segment name is used by the tool in its operation forms to identify the output segments.

PROCEDURE Statement

Statement Syntax:

```
PROCEDURE procedure_name [RETURN type] ;
```

The PROCEDURE statement begins a procedure section. It assigns an identifying name to the procedure and defines the return type, if the function returns a value. The procedure name is limited to 16 characters maximum.

BEGIN/END Statement

Statement Syntax:

```
BEGIN
   statements
END;
```

The BEGIN/END statement delineates the *body* of a template section. Recall that a section *body* contains DGL statements to be executed by the program. You may included any number of statements between `BEGIN` and `END`.

Comment Statement

Statement Syntax:

```
-- free text
```

DGL may include programmer comment lines in the template. These are lines of free text that are not interpreted or handled by the Documentor during execution. Such comments are useful for documentation purposes.

Comments are preceded by two dashes. The comment symbol may start anywhere in a line, except within a literal string. All characters after the comment symbol until the end of the line are ignored.

An example of a comment:

```
-- This line contains program comments
```

Declaration Statements

You declare identifiers and their data-types in declaration statements in the declaration part of each template section.

Declarations for global identifiers and parameters are made in the initiation section - these identifiers can then be used throughout the template.

Declarations for local identifiers, i.e., identifiers to be used only in the relevant segment, are made after each segment identifier line. Procedures start with declarations of parameters and local identifiers.

PARAMETER Statement

Statement Syntax:

```
PARAMETER data-type identifier [:= value] , ... ;
```

For Example:

```
PARAMETER STRING activity_name ;
```

There are “template parameters” for the entire template and “procedure parameters” for procedures. Template parameters are variables whose value may be changed interactively when the template is executed.

Declaration of template parameters is allowed only in the initiation section. The keyword PARAMETER appears only once in the declaration section, before the data-type assignments for parameters. Each data-type statement may be followed by as many identifiers of the same type as you want to define.

For Example:

```
PARAMETER STRING activity_name, state_name, event_name;
```

As many type statements as desired may follow the PARAMETER keyword.

For Example:

```
PARAMETER
  STRING activity_name;
  FLOAT a:=3.243;
```

Template parameters may **not** hold a Rational Statemate element, while it is legal for a procedure parameter to be of this type. Procedure parameters are In/Out parameters.

Value assignments for PARAMETER statements are optional and allowed only for template parameters (we made one such assignment in the above example). If it is assigned, it represents the default value of the parameter at the first generation of a particular document. The value may only be a *literal* constant, not a constant identifier or an expression.

Note

Avoid changing template parameters within the template. It may cause confusion and can create inconsistent results when parameters are changed within segments.

CONSTANT Statement

Statement Syntax:

```
CONSTANT data-type identifier := value , ... ;
```

Constants are identifiers that have a defined value that cannot be changed in DGL statements.

The keyword CONSTANT appears only once in the declaration section, before the data-type assignments for constants. Each data-type statement may be followed by as many identifiers of the same type as you want to define.

For Example:

```
CONSTANT integer a:=1, b:=2, c:=3;
```

As many type statements as desired may follow the CONSTANT keyword.

For Example:

```
CONSTANT
   STRING activity_name:='Print';
   FLOAT a:=3.243;
   INTEGER c:=6;
```

The constant type may **not** be a *Statemate element*, or a *list of type*.

The identifiers in the CONSTANT statement must have their values assigned in the statement. The value may be any expression not containing variables or parameters.

VARIABLE Statement

Statement Syntax:

```
VARIABLE data-type identifier [:= value],... ;
```

Variables are identifiers whose values may be changed in other DGL statements.

The keyword VARIABLE appears only once in the declaration section, before the data-type assignments for variables. Each data-type statement may be followed by as many identifiers of the same type as you wish to define.

For Example:

```
VARIABLE STRING act_name, act_syn, act_desc;
```

As many type statements as desired may follow the VARIABLE keyword. For example:

```
VARIABLE
   string activity_name;
   float a:=3.243;
   activity act_id;
```

Value assignments are optional (we made one such assignment in the above example). If they are assigned, they represent the default value of the variable at the first generation of a particular document. The value may be any expression that does not contain other variables or parameters.

Variables that are declared as *Statemate elements* and *list of* items may not be assigned initial values.

Assignment Statement

Statement Syntax:

```
variable := expression;
```

This statement should be interpreted as follows: the variable on the left-hand side of the statement is assigned the value of the expression on the right-hand side.

The expression and the variable must be of the same or compatible type. If the expression is of type STATE, ACTIVITY, etc., the variable is of the same type, or of type ELEMENT.

Here is an example of a template with a declaration section, followed by a section containing an assignment statement.

```
VARIABLE
   LIST OF STATE      st_list;
   LIST OF ACTIVITY   act_list;
   LIST OF ELEMENT    el_list;
BEGIN
              .
              .
  el_list := st_list + act_list;
              .
              .
END;
```

File Handling Statements

These statements open and close files or the dialog area, to and from which you can pass text or messages, through the use of WRITE and READ statements. The READ statement is also included in the file handling statements, whereas the WRITE is considered to be an output statement.

OPEN Statement

Statement Syntax:

```
OPEN (fl, file_name, mode [, status]);
```

where `fl` is an identifier of type FILE and `mode` is either INPUT or OUTPUT.

This statement is used with `mode`=OUTPUT to open a file or the dialog area so that a subsequent WRITE statement can pass text to it, and with `mode`=INPUT for subsequent READ statements. The statement assigns a value to `fl`.

The dialog area is frequently opened to pass run-time messages to it. To open the dialog area, use the string 'DIALOG' (with the apostrophes) for the `file_name`.

Optionally, you may include the status function code `status` which returns the value *stm_success* upon successful execution of the statement.

If the file opened for output does not exist, this statement creates a new file.

If the file exists, it is initialized by the OPEN statement; i.e., the written information overwrites the existing contents of the file.

CLOSE Statement

Statement Syntax:

```
CLOSE (fl);
```

where `fl` is an identifier of type FILE. This statement closes a file that was previously opened with the OPEN statement.

READ Statement

Statement Syntax:

```
READ(fl, variable1, variable2, ...) ;
```

where `fl` is an identifier of type FILE that points to a file that was opened using an OPEN statement, in INPUT mode. The variables are identifiers of type *integer, float,* or *string.*

Each READ statement reads a line from the file. The numeric elements in the input line are separated by blank or tabs. Reading to a string variable reads the rest of the line.

For Example:

```
READ(fd, i, str)
```

where `i` is an integer and `str` is a string. This statement, when applied to an input line: `12 May 1991`, results in: `i`=12, `str`='May 1991'.

The READ statement may operate as a function that returns either *stm_success*, *stm_cannot_read_file*, or *stm_end_of_file*.

Output Statements

Output statements pass text to the output segment files. The text can originate from a number of sources: the template itself, user text files, database retrieval functions, Rational Statemate reports and plots, external programs, and evaluated expressions. Output statements cannot be put in the initiation section.

Below, we detail the various types of output statements.

Verbatim Statement

Statement Syntax:

```
/@ verbatim text @/
```

When the verbatim symbols `/@` and `@/` frame text in the template file, the text is passed literally (without interpretation) to the output segment file. Comments inside the frame, rather than being ignored, are passed literally as well. The end-of-statement character, ";", is not required following the concluding verbatim symbol.

Verbatim statements may be used to pass the following to the output file:

- Formatting commands applicable to a specific formatter.
- Short text passages such as titles, opening remarks, etc.
  In spite of the absence of any length restriction on verbatim text, longer text passages are usually passed using the INCLUDE file statement.

For example, when this verbatim section of the template is executed:

```
SEGMENT section1;
BEGIN      /@
.title ACTIVITY-SPEC
.skip 2
.center; AN ACTIVITY SPECIFICATION
-- This section will describe the
-- purpose of the activities.
@/
-- this line will not appear in the output
END ;
```

it results in the following output:

```
title ACTIVITY-SPEC
.skip 2
.center; AN ACTIVITY SPECIFICATION
 -- This section will describe the
 -- purpose of the activities.
```

WRITE Statement

Statement Syntax:

```
WRITE ( [fl,] write_expression , ... ) ;
```

The WRITE statement to write expression values to any of the following

- The document output segment
- Another file
- The dialog area of the tool window

You may write a numeric or string expression that is evaluated in the template, or a literal piece of text. The WRITE statement can also be used to write information retrieved from the database, such as element names.

Using the WRITE statement to write to a file or to the dialog area is particularly useful if you want to write messages (error messages, run-time messages, etc.). When writing to a file or to the dialog area, you must include the `fl` identifier. In such cases you must also precede the WRITE statement with an OPEN statement.

The WRITE statement is commonly used to write lines that include text (string literals) together with expression values.

For example: `WRITE('NAME:', di_name);`
results in the following being written in the output segment file NAME: KUKU , where *KUKU* is a value of `di_name`.

From the examples you can see that there can be more than one write expression. When there are multiple expressions they are separated by commas.

Lines of pure text are more suitably handled using the Verbatim statement.

Literal strings may include the formatting characters:

`\n` - new-line
`\t` - tab

For example, `WRITE('\n',alpha);`
would write the value of alpha at the beginning of the next line in the output segment file.

Optionally, you can specify the minimum number of characters to be written in the output file. You do this by using the following syntax for the write expression:

```
expression : num
```

where `expression` can be either a numeric or a string expression, and `num` is an integer constant or integer expression that represents the minimum number of characters that `expression` will occupy. `expression` and `num` may involve operands, operations, and function calls.

For Example:

```
WRITE(act_name: 10, ',' , act_synonym);
```

results in the string value for act_name being written in the output file to a length of at least 10 characters; if the name has less than this number, blanks are added to achieve the specified string length - as in:

```
COMP     , SET
```

In this example, spaces have been added to "COMP" to give it a length of 10 characters.

The use of num determines the minimum number of characters to be written in the output file, as follows:

- For a string, the length of the string is the minimum number of output characters. When specified, and where num is greater than the string length, blanks are padded to the right of the string to achieve a total string length of num.
- For an integer, when num is specified, and where num is greater than the number of digits in the integer, blanks are padded to the left of the number to achieve a total output length of num.
- For a real number, when num is **not** specified, the value is output to no more than 8 decimal places. Thereafter, the number is automatically rounded. When specified, and where num is greater than the digits output according to the default above, blanks are padded to the left of the number to achieve a total output length of num. Where num is less than the digits output according to the default above, the decimal portion of the number may be rounded to arrive at a specified output length of num. However, in no case will the integer portion of the real number be truncated.

For Example:

```
WRITE ('Name:':8, name, '\n', 'Value:':8, v) ;
```

Assume that name contains "Xfactor" and v is an integer that equals 5105. This writes the following lines to the output file:

```
Name:   Xfactor
Value:  5105
```

Note

The WRITE command may not access a *list* variable directly; if you attempt to write a variable which refers to a list, an error message is displayed. To output a list, use a control flow construct such as a loop - writing one list item at a time.

Using WRITE to Produce Messages

A WRITE statement may also be used to write information to a file or to the dialog area, instead of to the document itself. For this, you must first open a file in OUTPUT mode using the OPEN statement.

To write a string message to the file use the following syntax:

```
WRITE (fl, write_expression);
```

where `fl` is the file pointer to which you want to write the messages.

INCLUDE Statement

Statement Syntax:

```
INCLUDE (file_description [, status]);
```

where `file_description` is either a *file-name* or an identifier of type FILE.

This statement copies the text of a specified file to an output segment file. The text is passed to the file verbatim and may contain formatting commands for the format processor that is used to produce the formatted document.

When the `file_description` is a *file_name*. It can be any string expression; it may be a literal string inside quotes (e.g. 'ABC') or an evaluated expression that produces a file name (e.g., a variable that contains a file name).

Note

The *file_name* may include the directory pathname of the file, using the file name conventions of the host operating system. If you do not specify the pathname, the Workarea is searched for the file.When the `file_description` is of type FILE, the file from which you are copying text must first be opened with an OPEN statement in OUTPUT mode.

When using the INCLUDE statement you may optionally include the status function code `status` which, upon successful execution of the statement, returns a value of *stm_success*.

EXECUTE Statement

Statement Syntax:

```
EXECUTE (calling_sequence);
```

This statement invokes a program that is external to Rational Statemate. The Documentor searches for the program name using the regular system search path. The tool then invokes the program and sends its standard output to the document's segment file.

The `calling_sequence` is a string expression containing the name of a program and any arguments. This is exactly the calling sequence used to invoke the program from the operating system.

The calling sequence can be a literal string in quotes or an expression that evaluates to a string.

For Example:

```
EXECUTE ('DATE') ;
```

calls the operating system function `DATE` and prints the date in the output file.

The EXECUTE statement function operates as a function that returns either *stm_success* or *stm_error*.

Include Reports Statement

Statement Syntax:

```
stm_rpt_report_name ( report_parameters ) ;
```

This statement generates and writes a Rational Statemate predefined report such as a Tree, Structure, Interface etc. into the output file. It activates the Reports Tool, generating the specified report with the given parameters.

For example, the statement:

```
stm_rpt_tree(list,5);
```

produces a Tree Report for the items in a list represented by the variable `list` to a depth of 5 in the hierarchy. The report is included in the output file.

The output from the Reports Tool contains formatting commands applicable to the format processor attached to the template. If no formatter is specified, the Reports Tool cannot be invoked and an error message is issued.

Each of the predefined reports is invoked for a list of elements. The input parameter that represents this list is denoted by a variable name that must be of type *list of* one of the Rational Statemate element types. This variable, along with all other identifier names used in the calling sequence, must be declared in an appropriate declaration section. For example:

```
VARIABLE
    LIST OF ACTIVITY ac_list;
```

The identifier `ac_list` may be assigned a list of activities and then be included in a statement that generates a Dictionary Report, like this:

```
stm_rpt_dictionary(ac_list,. . . );
```

The report is generated for each item in the list represented by the `ac_list` variable.

A number of arguments are used to define the parameters for each report. Some are called *single character string arguments*. These are used to indicate restricted parameter choices. For example, in the Interface Report statement

```
stm_rpt_interface(elist, 'A', ...);
```

the value of the second argument `'A'` indicates that the Interface Report should be of type *activities*; specifying an `'M'` for this parameter would indicate that the report should be generated for *modules*.

The single character string arguments can be more than one character, but only the first character of the string is actually passed to the Reports Tool. If a non-valid character is passed to the Reports Tool, the report is not generated, and an error status code is returned.

Some of the arguments are Boolean and are evaluated as true or false to indicate whether or not some parameter is set. For example, in the Dictionary Report statement `stm_rpt_dictionary (elist, true, ...);` the Boolean constant `true` indicates that the *long description* will be included in the report.

The following a list of the Rational Statemate reports that can be invoked, together with their calling name, and required parameters.

- **Dictionary Report**

  ```
  stm_rpt_dictionary(elist,ldes,attr,attr_title) ;
  ```

  `elist` is a list expression of Rational Statemate elements for which the report is produced.

  `ldes` is a Boolean expression indicating whether or not the long description of each element should be included in the report.

  `attr` is a Boolean expression indicating whether the attributes of an element are included in the report.

  `attr_title` is a string indicating the attribute names whose value will precede the element name in the report.

- **Tree Report**

  ```
  stm_rpt_tree(elist, depth) ;
  ```

  `elist` is a list expression of Rational Statemate elements for which the report is produced.

  `depth` is an integer argument indicating to what hierarchical level the report should be generated. For all levels, enter the number “99”.

- **Protocol Report**

  ```
  stm_rpt_protocol(elist, attr_title) ;
  ```

  `elist` is a list expression of Rational Statemate elements for which the report is produced.

  `attr_title` is a string indicating the attribute name whose value will precede the element name in the report.

- **List Report**

  ```
  stm_rpt_list(elist) ;
  ```

  `elist` is a list expression of Rational Statemate elements for which the report is produced.

- **Structure Report**

  ```
  stm_rpt_structure(elist, width) ;
  ```

  `elist` is a list expression of Statemate elements for which the report is produced.

`width` is an integer argument indicating the page width (in characters) to be used for the report.

- **Attribute Report**

```
stm_rpt_attribute(elist, attrs, attr_title) ;
```

`elist` is a list expression of Rational Statemate elements for which the report is produced.

`attrs` is a list of strings that contains the specific attribute names for which the report should be generated. If this list is empty, then the report retrieves all the attributes for each element.

`attr_title` is a string indicating that the attribute value will precede its element name in the report.

- **Interface Report**

```
stm_rpt_interface(elist, rtype, chart, lact
lmod, ftype, dis, names) ;
```

`elist` is a list expression, that must be of the type *list of modules*, for which the report is produced.

`rtype` is a single character string argument indicating the report type:

  – *'A'* indicates activity interface report.
  – *'M'* indicates module interface report.
  – *'I'* indicates information interface report.

`chart` is a single character string argument indicating which arrows are taken into account when the report is generated:

  – *'A'* indicates activity-chart arrows.
  – *'M'* indicates the module-chart arrows.

`lact` is an argument of type *list of activities* indicating which activities are taken into account when the report is generated. If `lact` is an empty list, the default is *all* activities implemented by the center module.

Note: If `chart` is *'M'*, then this parameter has no function. `lact` must still be supplied here. For simplicity, the null list (`null`) may be used.

`lmod` is an argument of type *list of modules* indicating the *side modules* that interface with the central module for which the report is to be generated. If `lmod` is empty, the default is *all* modules except the center module's own ancestors and descendants.

`ftype` is a single character string argument indicating the kind of information flow to appear in the report:

- *'D'* indicates data-flows.
- *'C'* indicates control-flows.
- *'B'* indicates both.

`dis` is a single character string argument indicating the kind of information to appear in the report:

- *'I'* indicates flow labels.
- *'P'* indicates parent information items.
- *'B'* indicates basic information items.

`names` is a single character string argument:

- *'N'* indicates that the *name* appears for elements that flow between the boxes.
- *'S'* indicates that the *synonym* appears for elements that flow between the boxes.

- **N2 Chart Report**

```
stm_rpt_n2chart(elist, names, level, env, chart,
dis, ftype)
```

`elist` is a list expression, which must be of the *type list of modules* or *list of activities*, specifying the elements in the diagonal.

`names` is a single character string argument:

- *'N'* indicates that *names* of the elements appear on the diagonal of the matrix.
- *'S'* indicates that *synonyms* of the elements appear on the diagonal of the matrix.

`level` is a single character string argument indicating what appears on the diagonal when both parent box and sub-box are in the list.

- *'B'* indicates that *sub-box* is placed on the diagonal of the matrix.
- *'P'* indicates that the *parent box* is placed on the diagonal of the matrix.

`env` is a Boolean expression; if true then the *environment* is added to the matrix.

`chart` is a single character string argument indicating which arrows are taken into account when the report is generated:

- *'A'* indicates activity-chart arrows.
- *'M'* indicates the module-chart arrows.

`dis` is a single character string argument indicating the kind of information to appear in the report:

- *'I'* indicates flow labels.
- *'P'* indicates information items.
- *'B'* indicates basic information items.

`ftype` is a single character string argument indicating the kind of information flow to appear in the report:

- *'D'* indicates data-flows.
- *'C'* indicates control-flows.
- *'B'* indicates both.

- **Resolution Report**

```
stm_rpt_resolution(clist,type)
```

`clist` is a list of charts. It determines the scope of the report.

`type` is stm_type of elements to include in the report. The `type` may be one of the following:

stm_textual, stm_graphical, stm_mixed, stm_state, stm_module, stm_activity, or stm_data_store.

Include Plots Statement

Statement Syntax:

```
stm_plot (plot_parameters) ;
```

This statement is used to include any chart in the document. It generates the specified plot with the indicated parameters (e.g., plot size, output device, etc.). The `plot_parameters` are specified in the order given below.

The output is designated for a particular device (one of the output devices defined in Rational Statemate). The destination of the plot output is specified by one of the parameters. If its destination is left unspecified, it is included as part of the output segment file. This is done by using an empty string for the output file parameter.

The plot function returns a status which can be one of: stm_success, stm_unknown_plotter, stm_can_not_open_file, stm_id_not_found, stm_id_out_of_range, stm_plot_failure, stm_illegal_parameter, stm_not_enough_memory, stm_empty_chart, stm_unresolved.

Whether you are plotting Statecharts, Activity-charts, Module-charts, or Block Diagrams, the parameters are the same:

```
stm_plot (id, file, width, height, with_label, with_name, with_note, device,
title_position, title, do_rotate, with_file_header, actual_height)
```

- **id** - is the ID number of a Rational Statemate chart to be plotted.
- **file** - is a STRING with the name of the file destination to which the plot is written. The operating system pathname conventions are followed. You may specify a full pathname to any directory for which you have write access. If a simple filename is specified, the plot is written to your Workarea. If the parameter is left empty (`''`), the plot is included as part of the output file.
- **width** - is a numeric argument of type FLOAT that indicates the maximum possible width of the plot (in inches).
- **height** - is a numeric argument of type FLOAT that indicates the maximum possible height of the plot (in inches).
- **with_label** - is a BOOLEAN parameter which indicates whether arrow labels are (TRUE) or are not (FALSE) printed in the plot.
- **with_name** - is a BOOLEAN parameter indicating whether box names are (TRUE) or are not (FALSE) printed in the plot.
- **with_note** - is a BOOLEAN parameter indicating whether notes are (TRUE) or are not (FALSE) printed in the plot.

  Note:

  - **device** is a STRING argument that indicates the plotting device. This may indicate a supported formatting language if the plot is to be handled by a formatting processing system that has its own graphics language. To configure a new plotter or printer (for example, a paper type), select **Utilities > Output Devices** from the main Rational Statemate menu.
  - Plots created using the Word format in the Output Device dialog are **RTF** files.
- **title_position** is a STRING parameter indicating where to place the plot title. This parameter accepts one of the following values:

  stm_plt_none - the title is not included.

  stm_plt_top - the title is placed a the top of the plot.

  stm_plt_bottom - the title is placed a the bottom of the plot.

- **title** is a STRING argument that specifies what title will be printed with the plot.
- **do_rotate** is a BOOLEAN parameter where TRUE indicates landscape and FALSE indicates portrait.
- `with_file_header` is a BOOLEAN parameter where TRUE indicates that a header is to be added at the beginning of the file. (Use this if you do not want the plot as part of the document.)
- **actual_height** is a numeric argument of type FLOAT that indicates the actual height (in inches) of the plotted output.

The following is an example of how a plot is generated in DGL. A template contains the following statements:

```
VARIABLE
 CHART            ch_id;
 INTEGER         status;
 FLOAT           real_ht;
            .
ch_id:= stm_r_ch('XL25',status);
stm_plot(ch_id,'/sam/p_xl25',5.0,7.0,true,true,false,
         'POSTSCRIPT',stm_plt_top,'SystemXL25',true,true          ,real_ht);
           .
```

This produces a plot for the chart XL25 that is limited to a maximum size of 5 inches by 7 inches, prints labels and box names and does not print notes.

Output to the file specified by the path `/sam/p_xl25`.

This file is in Postscript format, as defined for device called 'POSTSCRIPT' in **Utilities > Output Devices** from the main Rational Statemate menu.

The file will have the appropriate header for the output device.

The plot will be printed in landscape orientation.

```
A title "system XL25 is printed at the top of the plot; the actual
height is returned by the variable real_ht.
```

Include Table Statement

Statement Syntax:

```
stm_table_simple (title, columns, contents, page_width, page_height, anchor);
```

This statement generates a simple table. You specify the number of columns and their width, and the information to be included in the table. The parameters are as follows:

- **title** is the title of the table. The title appears centered over the table.
- **columns** is a list of integers that specify the width of each column in number of characters. For instance: `{16} & {16} & {20}` specifies a table having three columns, the first two columns being 16 characters wide and the last column being 20 characters wide.
- **contents** is a list of strings containing the information to be entered into each cell of the table. The information fills the table horizontally, row by row, depending on how many columns were specified. For instance, if you specified three columns in the `columns` parameter, the following `contents`:

  ```
  {'Project Name'} & {'Date'} & {'Location'} &
  {'Alpha'} &{'April 1987'} & {'Boston'}
  ```

  would produce this table:

Project Name	Date	Location
Alpha	April 1987	Boston

- **page width** determines the width of the page in inches. It is only relevant for Interleaf - for other systems, you may specify `0.0` for this parameter.
- **page_height** determines the height of the page in inches. It is only relevant for Interleaf - for other systems, you may specify `0.0` for this parameter.
- **anchor**, relevant only for Interleaf. A character string indicates where to place the table. The options are 'A' - at anchor, 'F' - following anchor. The default is 'F'.

For formatters other than Interleaf, precede the table with your system's *no fill* and *no adjust* formatting commands.

The following is an example of how a table can be generated using function calls. Notice how we repeatedly assign new values to the `List_str` variable to build the table.

Also note that the first statement uses the NROFF commands for *no fill* (`.nf`) and *no adjust* (`.na`). These commands cause the word processor to take the text "as is." Some word processors see this mode as "verbatim" or "literal." The last statement uses the NROFF commands (`.fi`) and (`.ad`) to return to fill and adjust modes.

```
WRITE ('\n.nf \n.na \n');

BEGIN
 List_str:={'ACTIVITY NAME'} & {'ID'} & {'LANGUAGE'};'
 act_list:=stm_r_ac_logical_desc_of_ac({act_chart},st);
 FOR act IN act_list LOOP
   List_str:=List_str & {stm_r_ac_name(act,st)};

   attr_list:=stm_r_ac_attr_val(act,'ID_NUMBER',st);

   IF (st = stm_success) THEN
      attr_val:=stm_list_first_element(attr_list,st);
      List_str:=List_str & {attr_val};
   ELSE
       List_str:=List_str & { 'N/A'};
   END IF;

  attr_list:=stm_r_ac_attr_val(act,'LANGUAGE',st);
   IF (st = stm_success) THEN
      attr_val:=stm_list_first_element(attr_list,st);
      List_str:=List_str & {attr_val};
   ELSE
        List_str:=List_str & {'N/A'};
   END IF;
 END LOOP;

 WRITE ('\n.nf\n.na\n');
 title := 'Table CC1. Simple Table Example';
 Col_list := {16} & {10} & {40};
 stm_table_simple(title,Col_list,List_str,pg_w,pg_h, 'A');
 WRITE ('\n.fi\n.ad\n');
END;
```

The formatted output:

ACTIVITY NAME	ID	LANGUAGE
FUNCTION1	AC1-A1.1	FORTRAN
FUNCTION2	AC1-A1.2	N/A
FUNCTION3	AC1-A1.5	ADA
FUNCTION4	AC1-A1.4	N/A

Control Flow Statements

Several control flow constructs provide you with options for conditional and iterating statement execution. These resemble constructs in other conventional programming languages.

IF/THEN/ELSE Statement

Statement Syntax:

```
    IF boolean_expression THEN
    statements
[ ELSE statements ]
    END IF ;
```

The IF/THEN/ELSE construct is used for conditional execution of DGL statements.

In this statement, the statements following the THEN (and before any ELSE) are executed if the `boolean_expression` evaluates to true. If it is evaluated to false, the statements following the ELSE are executed, when present.

Here is an example:

```
IF a >= b THEN
     EXECUTE ('DATE') ;
     INCLUDE ('sample.txt') ;
  ELSE
     WRITE ('a is less than b') ;
END IF ;
```

SELECT/WHEN Statement

The SELECT/WHEN construct is used for conditional execution of DGL statements. This statement is more powerful than the previous IF/THEN/ELSE statement, in that it allows you to systematically list multiple conditions for statement execution.

Statement Syntax:

```
SELECT [selection_mode]

    WHEN trigger => statements
  [ WHEN trigger => statements ]
          .
          .
  [ WHEN ANY => statements ]
          .
  [ WHEN trigger => statements ]
          .
          .
  [ WHEN ANY => statements ]
          .
          .
  [ OTHERWISE => statements ]

END SELECT ;
```

The optional `selection_mode` can be either the keyword `FIRST` or `ANY`. The selection-mode determines the way the statements are checked for possible execution - this will be explained shortly. The default selection-mode is `FIRST`.

Note that WHEN statements are composed of two parts: the `trigger` to the left of the arrow, and `statements` on the right side of the arrow. The trigger is any valid Boolean expression. The statements following a trigger are performed only when the trigger is true. Whether or not these statements are actually executed also depends upon the selection-mode, as follows:

- If the selection-mode is `ANY`, then the statements are executed whenever their corresponding trigger is true.
- If the mode is `FIRST` (or not given), then only the first true trigger in the entire SELECT construct is executed; the rest are ignored, regardless of whether their triggers are true or not.
- The `WHEN ANY` statements are executed when one or more of the preceding `WHEN` statements have been executed.
- The `OTHERWISE` statements are executed only if no `WHEN` statement within the SELECT construct is triggered.

To demonstrate the execution of the SELECT/WHEN construct, consider the following example. `a, b`, and `c` are numeric variables.

```
SELECT ANY
  WHEN a = 5 => b := 10 ;
  WHEN a > b => b := 10 ;
  WHEN a = 0 => b := 0 ;
  WHEN ANY  => write ('a may influence b') ;
  WHEN c = 5 => b := 5 ;
  WHEN c > b => b := 5 ;
  WHEN c = 0 => b := 0 ;
  WHEN ANY  => write ('c may influence b') ;
  OTHERWISE => write ('b has not been changed') ;
END SELECT;
```

The execution is determined by:

1. Each `WHEN` statement is triggered if its corresponding expression is evaluated to true.
2. The first `WHEN ANY` statement is triggered if `a` is equal to 5, greater than `b`, or equal to zero.
3. The second `WHEN ANY` statement is triggered if at least one of these same conditions is true with respect to the variable `c` instead of `a`.
4. The `OTHERWISE` statement is triggered only if the value of `b` has not been changed within the SELECT statement's evaluation.

When processing a `WHEN ANY` statement, the Documentor Tool only "looks back" to the previous WHEN ANY construct (if one exists). Therefore, in the above example, if `a = 0` and none of the tests of `c` were true, the WRITE statement's message `c may influence b` is not issued.

In this example, what would happen if the selection mode was FIRST instead of ANY, and the conditions a > b and c = 5 were both true ?

In this case the assignment b: = 10 and the first write message (the corresponding WHEN ANY statement) are executed. The assignment of b to 5 along with its corresponding WHEN ANY statement are not done because c = 5 is not the *first* true trigger.

The statements following the => symbol in the WHEN constructs may be any valid DGL statements. You may even enter a SELECT construct at this point. This allows you to nest SELECT constructs. There is no limit to the depth of nested SELECT blocks.

FOR/LOOP Statement

Statement Syntax:

```
FOR identifier IN list LOOP
   statements
END LOOP ;
```

The FOR/LOOP construct is used for iterative execution of DGL statements. The statements after the keyword LOOP are executed for each element in the specified list.

Alternatively, a range of integers can be specified in place of the list, as in the following example:

```
FOR i IN {1..100} LOOP
   statement ;
     .
     .
     .
END LOOP ;
```

The identifier is a variable whose value is set sequentially to the items in the list. The type of the identifier must match the type of the list. This variable may be used within the body of the loop.

The following example writes the name and synonym for each state found in the list defined by the variable sub_states:

```
VARIABLE
  STATE           st_id ;
  LIST OF STATE   sub_states;
          .
          .
          .
  FOR st_id IN sub_states LOOP
    WRITE(stm_r_st_name(st_id,status), '   ',
        stm_r_st_synonym(st_id,status), '\n') ;
  END LOOP ;
```

WHILE/LOOP Statement

Statement Syntax:

```
WHILE boolean_expression LOOP
   statements
END LOOP ;
```

The WHILE/LOOP construct is used to iteratively execute DGL statements. The execution of `statements` is determined by evaluation of the `boolean_expression`.

The statements are executed until the expression evaluates to false. For example,

```
WHILE  a > b  LOOP
          .
          .
     b := b + k ;
          .
          .
END LOOP ;
```

The statements between the keywords `LOOP` and `END LOOP` are executed as long as `a` is greater than `b`.

Assume that `b` changes its value inside the `loop` and in one of the iterations the expression `a > b` becomes false. In the next iteration, the expression is examined and, since `a > b` is now false, the execution of template statements continues with the first statement after the `END LOOP`.

EXIT Statement

Statement Syntax:

```
EXIT ;
```

The EXIT statement is relevant only inside FOR or WHILE loops. It results in an exit from the current loop to the statement after the loop construct or to the construct that contains the current loop.

Typically, a condition would be tested in a loop, and the exit would be based upon the evaluation of that condition.

For example, the following structure would continue the execution of the statements between the `LOOP` and `END LOOP`, depending on the value of the condition `a > b`.

The iterations go on as long as the status `st` is equal to `0`. When `st` is not equal to `stm_success`, it causes the iterations to stop. The `IF` statement here is used to force an abnormal `EXIT` from within the `WHILE` loop.

The execution resumes at the next statement after the `END LOOP`.

```
WHILE a > b LOOP
   md_id := stm_r_md (name , st) ;
   IF st <> stm_success THEN
       WRITE (’Illegal Status’) ;
       EXIT ;
   END IF ;
       .
       .
   END LOOP ;
```

STOP Statement

Statement Syntax:

```
STOP ;
```

The STOP statement stops execution of the template.

Typically, a specific condition is tested and the template is stopped if this condition has a value for which further processing is meaningless.

For example, in the following statements we check whether the specified `system_name` is proper. If not, i.e., if an error has been detected, a message is issued to the dialog area and the template is stopped.

```
md := stm_r_md (system_name, status) ;
IF (status <> stm_success) THEN
   WRITE (dialog_area, ’Execution Stopped due                          to
error’) ;
   STOP ;
END IF ;
```

Documentor Functions

When you write reports on system design specifications, it is often necessary to include information from the Rational Statemate database that contains the specifications. The Documentor provides you with this capability through the use of database extraction functions.

Overview of the Extraction Functions

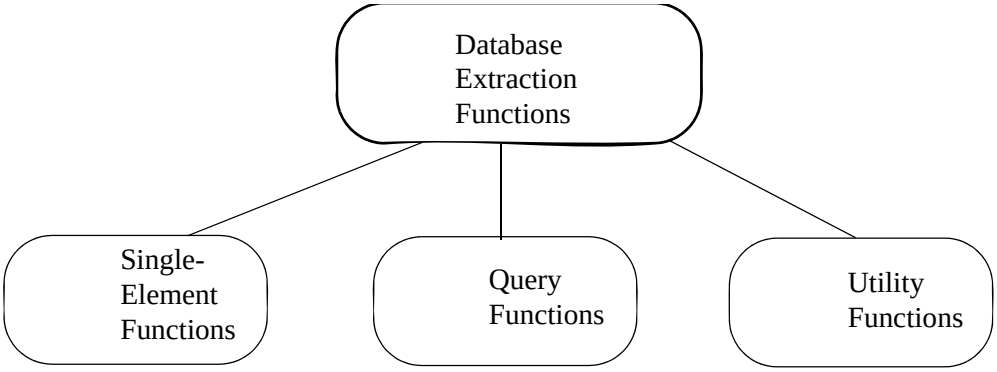
The database extraction functions are a collection of routines that enable you to extract information from the specification database.

There are four types of database extraction functions:

- **Single-element** - Provide information about a discrete Rational Statemate element in the specification database. For example, you can retrieve the contents of the description field in the form for a particular state. See **Single-Element Functions** for detailed information.
- **Query** - Extract lists of elements from the database that conform to a specific criterion. For example, you can extract a list of activities from the database that are control activities. Each function corresponds directly to a query of the property sheet. Function output consists of a list of Rational Statemate elements. See **Query Functions** for detailed information.
- **Utility** - Perform operations on lists and strings. Most of these functions do not extract information from the database, but enable you to manipulate the information you have already retrieved. See **Utility Functions** for detailed information.
- **Project management** - Extract information about the Statemate project, manager, and members.

Function Structure

The following figure shows the structure of the database extraction functions.



There are scores of database extraction functions. However, it is easy to become proficient in their use because of the systematic structure of the package. The functions are documented in **Single-Element Functions**.

Using Database Extraction Functions

Database extraction function calls can appear anywhere in your template where expressions of the same type are valid.

Consider the following call:

```
state_id := stm_r_st ('S1', status);
```

This call retrieves the state whose name is `S1` from the database and assigns it to the variable `state_id`. (In actuality, the call retrieves the state's ID. This ID is a value that Rational Statemate uses to identify each element in the database.)

Function calls are frequently used in sequence. For example, because you have already retrieved the ID for state `S1`, you can now call the following function:

```
sub_st := stm_r_st_physical_sub_of_st
   ({state_id}, status);
```

This function call builds a list of substates contained in state `S1` and assigns the list to the variable `sub_st`.

At this point, you can print out a list of all substates of state `S1`. The list is to include the name of the individual state as well as the description appearing in the state's form. To do this, include the following lines in your template:

```
FOR s in sub_st LOOP

WRITE ('\n Name:', stm_r_st_name (s, status));
WRITE ('\n Desc:', stm_r_st_description (s, status));
END LOOP;
```

Calling Conventions

Database extraction functions provide you with information from the Rational Statemate database. To extract this information, you call the specific function that retrieves the information you want. You specify the particular Rational Statemate elements that interest you as input arguments to the function. The function returns the information and a status code (as an output argument). This status code informs you whether your function call was successful.

Function Names

Single-element and query functions use the following prefix:

```
stm_r_
```

This prefix designates the function as a Rational Statemate database retrieval function.

Utility functions use the following prefix:

```
stm_
```

Element Type Abbreviations

Database extraction functions use two-character abbreviations to identify the type of Rational Statemate elements referenced in function calls. The following table lists the element types and their abbreviations.

Element	Abbreviation
A-flow-lines (basic)	ba
A-flow-lines (compound)	af
A-flow-lines (local)	laf
Actions	an
Actors	actor
Activities	ac
Boundary boxes	bb

Element	Abbreviation
Charts	ch
Combinational assignments	ca
Conditions	co
Connectors	cn
Data-items	di
Data-stores	ds
Enumerated value	en
Events	ev
Fields	fd
Information-flows	if
Local data	ld
Messages	msg
Mixed (multiple types)	mx
M-flow-lines (basic)	bm
M-flow-lines (compound)	mf
M-flow-lines (local)	lmf
Modules	md
Module-occurrences	om
Off-page activities	oa
Routers	router
Separators	sep
States	st
Subroutines	sb
Subroutine parameters	sp
Timing constraints	tc
Transitions (basic)	bt
Transitions (compound)	tr
Use cases	uc
User-defined types	dt

For example, `stm_r_ac_name` retrieves the name of an activity, whereas `stm_r_st_name` retrieves the name of a state.

The naming structure for each type of database extraction function is explained in the section that describes each specific type. Note that element type and the information to be extracted are contained in the function name and are *not* passed as arguments.

Arrow Elements

Arrow elements (transitions, a-flow-lines, and m-flow-lines) can be either basic or compound:

- A basic arrow connects a box or connector to another box or connector.
- Compound arrows are the logical connection between boxes and can possibly be composed of several basic arrows, passing through their connectors.

Function Input Arguments

Database extraction functions require input arguments in order to locate Rational Statemate elements in the database. Input arguments consist of elements or lists of elements for which information is sought.

Some functions require additional input arguments. Each argument must be declared to be of a data type recognized by DGL. This guide includes a complete list of input arguments for each type of database extraction function in the sections that describe the specific function type.

See the function reference appendixes for the lists of arguments relevant for each function.

Status Codes

Database extraction functions return only one argument—the function status code. This code reports whether the function call was successfully completed. When the function call fails, the status code indicates the problem. You can use the status code to pinpoint run-time errors in your template. For example, assume the following call appears in your template:

```
state_id := stm_r_st ('%', status);
```

The function requires a state name for the first (input) argument. In this case, the function returns a status code of 3, `stm_illegal_name`, because `%` is not a valid element name.

The status code is an integer value. Therefore, the `status` argument must be a variable declared as `INTEGER`. The Documentor provides predefined constants for the function status codes. This enables you to use the status name attached to each status code in your template.

For example, assume that you want to print out the synonym of the state `S1`. If there is no synonym defined in the state's form, print "`missing synonym`". Your template should contain the following code:

```
VARIABLE
    INTEGER        status;
                       .
                       .
                       .
state_id := stm_r_st ('S1', status);
synonym  :=  stm_r_st_synonym (state_id, status);
IF status = stm_missing_synonym THEN
   WRITE ( '\n synonym:    *missing synonym* ');
ELSE
   WRITE ( '\n synonym:    ', synonym);
END IF;
```

Status codes have severity levels that you can check to ensure that your function call was successful. These severity levels, and a complete list of status codes are documented in **Function Status Codes**.

Function Return Values

Database extraction functions return values that are of DGL data types (see **Data-types** for more information). Different functions return different types of values. For example, a function that retrieves the name of a Rational Statemate element returns a value of type `STRING`, whereas a function that retrieves a state's ID returns a value of type `STATE` (or `ELEMENT`).

The return value data type must be valid when the function is used in statements and expressions. For example, a return value that is a `STRING` can appear in places where string expressions are allowed, as shown in the following `WRITE` statement:

```
WRITE (stm_r_st_name (st_id, status));
```

This statement prints out the name of the state whose ID is `st_id`. The return values of each function are listed in the sections that describe the specific functions.

Return Values of Type ELEMENT

There are a number of database extraction functions that retrieve elements or list of elements from the specification database. In the case of elements, the functions return values that belong to DGL data types `STATE`, `EVENT`, `ELEMENT`, and so on. For example, consider the following function call:

```
state_id := stm_r_st ('S1', status);
```

This call extracts the state `S1` from the database. Because the function returns a `STATE`, `state_id` must be declared to be of type `STATE` or `ELEMENT`.

In the case of list of elements, the functions return values that belong to the DGL data types `LIST OF STATE`, `LIST OF ACTION`, `LIST OF ELEMENT`, and so on. All query functions return a list of Rational Statemate elements.

Return Values of Filename

A number of database extraction functions store extracted information in files, such as a function that retrieves an element's long description. In this case, the function returns the name of the file that contains the requested information. The filename returned is of DGL type `STRING`.

Return Values of Enumerated Types

DGL does not directly support enumerated data types. Functions that return discrete numerical values are considered to be of type `INTEGER`. The Documentor enables you to reference these numerical values by name. In reality, these names are internally defined as predefined constants in DGL. These names contain the prefix `stm_`.

For example, the function `stm_r_st_type` extracts the type of state specified in the function call. The possible state types are `stm_st_and`, `stm_st_or`, `stm_st_diagram`, and `stm_st_component`. These correspond to the values 0, 1, 2, 3 (respectively).

You can use the value names in your template. For example:

```
IF stm_r_st_type (st_id, status) = stm_st_component
THEN
       ...
```

The possible values that functions return, and their names, are documented in the function reference sections.

Model Templates

This section provides complete examples of templates—two that do not contain commands for a formatter (formatting is accomplished through DGL `WRITE` statements) and two that contain formatting commands for specific formatters.

The templates are as follows:

- **Properties**
- **Activity Interface Report**
- **Template for nroff**
- **Template for Interleaf**

Properties

This template uses database extraction functions to generate properties. It produces the property report for a list of data-items whose names match the specified pattern.

Each entry includes the name of the data-item, a short description, synonym, type (real, integer, and so on), and attributes and their values. In addition, it includes specific information extracted from each data-item's long description.

The template is as follows:

```
TEMPLATE di_dict;
-- this template produces information for each
-- data-item in a list no text formatter is used.
-- Formatting is done in WRITE statements.

PARAMETER
   STRING di_pattern; -- the pattern that
           -- determines the data-item list

VARIABLE
   INTEGER   st;          -- return status code
   LIST OF DATA_ITEM di_ids; -- list of
            -- data-items for which the report
            -- is produced

BEGIN
   di_ids:= stm_r_di_name_of_di (di_pattern, st);
END;
```

```
SEGMENT report;

VARIABLE
   DATA_ITEM          di;
   STRING             di_name, di_synonym;
   STRING             di_sdesc;
   INTEGER            di_type;
   STRING             di_desc_file;
   LIST OF STRING     attr_list, attr_val_list;
   STRING               attr, attr_val;

BEGIN

-- write report heading
WRITE ('\n\n',' ':20, 'DATA-ITEM PROPERTY');

FOR di IN di_ids LOOP

   di_name := stm_r_di_name (di, st);

   -- write entry heading
   WRITE ('\n\n\n   DATA-ITEM: ', di_name);
   -- get short description and print it out
   di_sdesc:= stm_r_di_description (di, st);
   WRITE ('\n\n', di_sdesc);
   -- get synonym and print it out
   di_synonym := stm_r_di_synonym (di, st);
   WRITE ('\n\n SYNONYM: ', di_synonym,'\n' );
   -- get structure\type and print it out
   di_type   := stm_r_di_data_type (di, st);

   SELECT
      WHEN di_type = stm_di_alias =>
         WRITE ('\n TYPE: ALIAS \n' );
      WHEN di_type = stm_di_constant =>
         WRITE ('\n TYPE: CONSTANT \n' );
      WHEN di_type = stm_di_primitive =>
         WRITE ('\n TYPE: PRIMITIVE \n' );
      WHEN di_type = stm_di_reference =>
         WRITE ('\n TYPE: REFERENCE \n' );
      WHEN di_type = stm_di_compound =>
         WRITE ('\n TYPE: COMPOUND \n' );

   END SELECT;

   --
   -- retrieve the attributes for the
   -- data-item of interest
   --
   attr_list := stm_r_di_attr_name (di, st);
   --
   -- write out the attribute names and values

   FOR attr IN attr_list LOOP
      attr_val_list:= stm_r_di_attr_val (di, attr,st);
      FOR attr_val IN attr_val_list LOOP
         WRITE ('/n', attr + ':':20, attr_val);
      END LOOP;
   END LOOP;
   --
   --
   -- write out various parts of the data-item's
   -- long description
```

```
   --
   di_desc_file := stm_r_di_keyword (di, '!PURPOSE',
                   '!END PURPOSE', '', st);
   WRITE ('\n\nPURPOSE: \n');

   IF st = stm_success
   THEN
      INCLUDE (di_desc_file);
   ELSE
      IF st = stm_starting_keyword_not_found
      THEN
         WRITE('  not available \n');
      END IF;

   END IF;

  di_desc_file := stm_r_di_keyword (di, '!TIMING',
                    '!END TIMING', '', st);
   WRITE ('\n\nTIMING: \n');

   IF st = stm_success
   THEN
      INCLUDE (di_desc_file);
   ELSE
      IF st = stm_starting_keyword_not_found
      THEN
         WRITE('  not available \n');
      END IF;
   END IF;

   di_desc_file:=
     stm_r_di_keyword (di, '!REPRESENTATION',
          ' !END REPRESENTATION', ' ', st);
   WRITE ('\n\nREPRESENTATION: \n');

   IF st = stm_success
   THEN
      INCLUDE (di_desc_file);
   ELSE
      IF st = stm_starting_keyword_not_found
      THEN
         WRITE ('  not available \n');
      END IF;
   END IF;
END LOOP;
END;
```

Properties Template Structure

As for all templates, this template is divided into two parts:

- An initiation section that contains global declarations and statements.
- Template segments, each of which contains local declarations and statements. Recall that each template segment, when executed, generates a document segment.

In this template, there is an initiation section and one segment. The initiation section follows the template identifier line:

```
TEMPLATE di_dict;.
```

The segment section follows the segment identifier line:

```
SEGMENT report;.
```

Both the initiation section and the template segments consist of two parts:

- **Declaration section** - Where the identifiers' types are declared.
- **Body** - Contains execution statements. These statements are contained within the keywords `BEGIN` and `END`.

Properties Initiation Section

The initiation section begins with the mandatory `TEMPLATE` identifier line that specifies the name of the template (in this case, `di_dict`). Following this are several optional comment lines describing the purpose and features of the template. As the comment lines state, this template does not contain formatting commands for any specific formatter.

Declaration Part

Identifier declarations follow the comment lines. There is one parameter, `di_pattern`, which is declared as a string and represents a string pattern. It will be used in a database extraction function to retrieve data-items whose names match this pattern. Because `di_pattern` is a parameter, you specify its value in a form before executing the template. Changing the string pattern enables you to generate the properties for different lists of data-items.

Next, the template declares two variables:

- `st` - Returns a status code used with all Statemate database extraction functions. It is declared as type `INTEGER`. See **Documentor Functions** for more information on using return status codes
- `di_ids` - Represents the list of IDs of data-items for which the report is produced. It is declared as a `LIST OF DATA_ITEM`.

The identifiers declared in the template so far belong to the initiation section and thus are global in scope (they can be used throughout the entire template).

Body

The body of the initiation section is between the `BEGIN` and `END` keywords. In this template, the body consists of only one statement, an assignment statement that assigns a value to the `LIST OF DATA_ITEM` variable, `di_ids`. A database extraction function assigns to `di_ids` a list of elements whose names match the string pattern `di_pattern`. These are the data-items for which the report will be generated. In this example, assume that you specified the string pattern `SIGNAL*`. The data-items for which the properties are generated will all have names that begin with `SIGNAL` (see the generated output in **Final Output for Data-item Properties**.

Note that the template uses a status code variable, `st`, as an argument in the function. However, the template does not include a statement to check `st`. If `st` does not indicate success, the returned list of data-items assigned to `di_ids` is empty.

Properties Segment Section

The long segment of the template begins with its mandatory identifier line, `SEGMENT report;`. This name is used to attach a name to the segment that follows. In templates possessing more than one segment, you can use this name to identify the specific segment. See **Using Documentor**.

The declaration part follows. First, it declares the variable `di` to represent a data-item. This variable will be used to iterate on the list of data-items.

Next, the template declares variables used in database functions to extract information about each data-item in the properties. The variables—`di_name`, `di_synonym`, and `di_desc`—correspond to the information contained in the name, synonym, and description fields of the data-item form shown previously. All three of these variables are declared to be of type `STRING`.

The next variable, `di_type`, represents the information entered in the type field of the data-item form. Because each of the options in this field corresponds to an integer value, this variable is declared to be of type `INTEGER`. Each integer value corresponds to a predefined constant; these constants, instead of the integer values, are used later in the template.

The string variable `di_desc_file` represents the name of a file. This variable will be used for the names of the files that include portions of the data-item's long description.

Next, the template includes variables having to do with information pertaining to the attribute fields. The variable `attr_list` represents the list of attribute names entered in the attribute name field of the data-item form. You will build this list later in the template through the use of a Statemate database extraction function. Similarly, the variable `attr_val_list` represents the list of attribute values. Lastly, the variables `attr` and `attr_val` are used to represent individual attribute names and values, respectively; these variables are used later to iterate on the list of attribute names and values.

Generating the Report Heading

The first line of the template body produces a report heading for the properties. To produce this heading, use a `WRITE` statement (preceded by a comment):

```
-- write report heading
   WRITE ('\n\n',' ':20, 'DATA-ITEM PROPERTY');
```

The character `\n` causes a blank line to be inserted in the text. In this case, two blank lines are inserted and the string "`DATA-ITEM PROPERTY`" is written, indented by blanks. The blank, `''` , followed by the number 20 that appears in the `WRITE` statement acts as a tab, indenting the line 20 spaces. In actuality, this number determines the minimum length that the printed expression is to reach. If the output falls short of this, blank spaces are added after the last character to complete the number. In this case, a single blank is used as the expression to be printed, causing 19 spaces to be added to the blank before the start of the next word.

Iteration: Using the FOR/LOOP Statement

The data-item properties will consist of information for more than one data-item—you want information for a *list* of data-items. Each data-item in the list should appear as a separate entry with the corresponding information (synonym, description, attributes, and so on). To generate the same type of information for all the data-items in the list, use a `FOR/LOOP` statement. This structure iteratively performs statements for a given identifier whose value is set sequentially to the items in a list. The program executes all statements following the `LOOP` keyword for the identifier's current value until it reaches the `END LOOP`; the program then loops back to the first statement in the loop and performs the same sequence of statements for the next item in the list.

In this example, the list is composed of data-items and is represented by the variable `di_ids`. The sequence of statements in the loop is generated for each item of the list, represented by the variable `di`. The loop terminates at the `END LOOP`, which is the second to the last statement in the template.

Generating the Entry Heading

Each data-item entry in the properties is marked by a heading that writes the words "`DATA-ITEM`:" followed by the name of the data-item for which the entry is made. The function `stm_r_di_name` extracts the name for the data-item represented by `di`. The name of the data-item entry is represented by the variable `di_name`.

To produce the entry heading, use a `WRITE` statement (preceded by a comment):

```
-- write entry heading
   WRITE ('\n\n\n   DATA-ITEM: ', di_name);
```

Extracting and Printing Information from the Data-Item Form

The next part of the template includes statements that extract and write information contained in the data-item form. This includes the data-item short description from the description field, synonym, data-item type, and attribute names and their values.

Description and Synonym

The first two statements in this part extract the short description of the data-item (the description field of the data-item form) and prints it out. The template uses a database extraction function to assign a value to the variable `di_sdesc` for the data-item represented by `di`. The `WRITE` statement that follows inserts a blank line, then writes the description (`di_sdesc`) of the data-item.

Next, two statements extract the data-item's synonym and prints it.

Using the SELECT/WHEN Construct

The next part of the template uses a `SELECT/WHEN` construct to extract and write the data-item type. To see how this works, first consider the fact that there are a finite number of values that can be specified for `type`. Turning to the `type` field in the data-item form, you can see that a data-item type can be one of the following:

- Record
- Integer
- Float
- String

The data-item type is referred to as an *enumerated type*—each of the values for this field is represented by a unique number. These numbers correspond to predefined constants (`stm_di_record`, `stm_di_list`, and so on). You use a database extraction function to extract this type for each data-item `di` in your properties.

Next, you want the name of the data-item type to be written in the document. To do this, use a `SELECT/WHEN` construct as shown in the template. This construct selects for execution one of the statements contained between the keywords `SELECT` and `END SELECT`. Only one of these statements is executed for each data-item. Each statement contains a trigger, beginning with the keyword `WHEN` that, when satisfied, leads to the execution of the statement following the arrow. In this template, the triggering condition is the value of the data-item type represented by the variable `di_type`. When this value is equal to that of the particular value of the data-item type contained in the `WHEN` statement, a `WRITE` statement is executed that inserts a blank line and writes "TYPE:", followed by the data-item type.

Note that you cannot write the data-item type directly from the variable `di_type` that represents it. This is because this variable actually represents an integer value; writing the variable's value would result in an integer being written, instead of the name of the data-item type. You solve this problem by writing an appropriate string name that corresponds to the variable's value.

Using Nested FOR Loops to Extract Attribute Names and Values

The next part of the template involves the use of nested `FOR/LOOP` constructs for extracting and writing the data-item's attribute name and value. A data-item's attributes and their values are recorded in special fields in the data-item form. The information in these fields is textual and can involve any features of the data-item that you want. The field enables you to record a list of attributes and values.

Recall that `FOR` loops are used to iterate on items of a list. In this template, the list of data-item attributes is represented by a `LIST OF STRING` variable, `attr_list`. The template executes a sequence of statements for each attribute (represented by the string variable `attr`) contained in the list.

These statements involve extracting and writing the corresponding attribute values for each attribute listed in the attribute name field. First, the template uses uses a database extraction function to assign values to a `LIST OF STRING` variable, `attr_val_list`. This variable represents the list of attribute values recorded in the attribute value field for a particular attribute name, represented by the variable `attr`.

This template assumes that more than one value can be recorded for each attribute; therefore, another `FOR/LOOP` construct is used to write the information for each value (represented by the variable `attr_val`) in the list.

You want to write attribute name and value as a paired set, with the attribute value written next to the attribute name. The template uses the number 20 in the `WRITE` statement to indent the second output field 20 characters. Note that for ":" to appear after the attribute name, the template uses a concatenation of two strings (`attr +':'`).

Using Keywords to Write Portions of the Long Description

In the last part of each property entry, you want to include information from the data-item's long description. This information consists of free text that is attached to the element's form. The long description can be divided into portions through the use of keywords. For example, you can set off those portions of the text that deal with data-item purpose by enclosing the text within the keywords `!PURPOSE` and `!END PURPOSE`. You can extract this information using a database extraction function that searches for keywords in the text and returns (to a file) all the text contained between these words. You can then use an `INCLUDE` statement to cause the file containing the text to appear in the document.

For each section of text that you want to appear, you begin by assigning a value to the string variable `di_desc_file`, which represents the name of the file with the desired text. Note that the fourth argument of the database extraction function is an empty string. This argument is reserved for the name of the file in which the desired text is returned. If you want the information in a specific file, you can specify a name for this argument. If you specify an empty string, as shown in the template, the Documentor writes the text in a temporary file. In either case, the function returns the name of the file in which the text is written. For example, in the first section, `di_desc_file` represents the temporary file that contains all the text between the keywords `!PURPOSE` and `!ENDPURPOSE`.

The assignment for `di_desc_file` is followed by an `IF/THEN/ELSE` statement that either includes the text or prints a "not available" message if the starting keyword is not found. This is done by means of the status parameter `st`, which appears in the keyword's extraction function. If the value of `st` is equal to `stm_success`, the text is included in the document. If the value of `st` is equal to `stm_starting_keyword_not_found`, the message "not available" is written. Any other status values will leave the paragraph empty.

The rest of the template extracts long description information in the same manner. The template execution then loops back to the beginning, repeating the instructions for each data-item in the list.

The final output is shown in the following section.

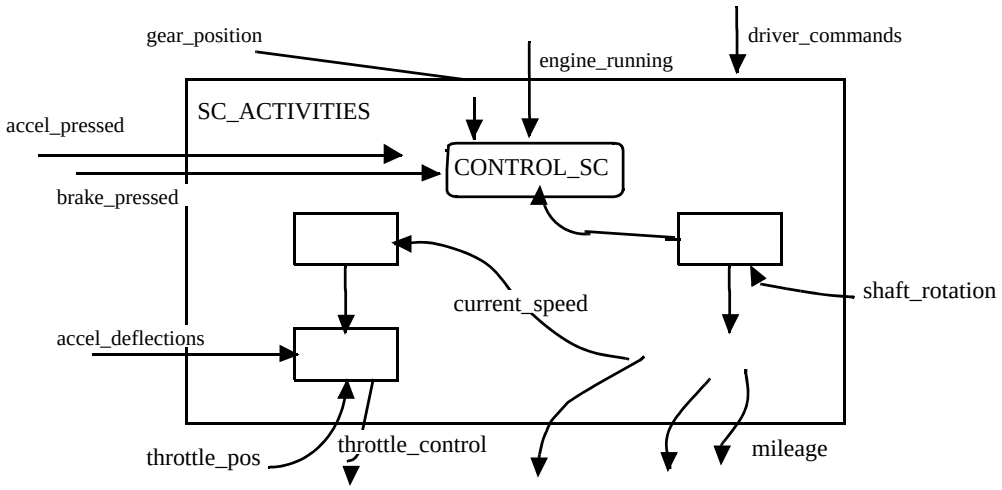
Final Output for Data-item Properties

```
                        DATA-ITEM PROPERTY

   DATA-ITEM: SIGNAL1

Signal specifying the current address

 SYNONYM: SIG_1

 TYPE: INTEGER

 RANGE:      0 - 2046
 RESOLUTION: 2

PURPOSE:
This signal designates the destination of the data that is sent to the system.
The system will transfer the signal to the component that is responsible to
the routing of information.

TIMING:
The address signal must be valid when the write signal is sent.

REPRESENTATION:
The signal uses the binary representation.

   DATA-ITEM: SIGNAL2

Signal specifying the data to write

 SYNONYM: SIG_2

 TYPE: INTEGER

 RANGE:       0 - 1000

PURPOSE:
This signal is the data that is sent to the system. The system will transfer
the data to its destination according to the specified address.

TIMING:
The data must be valid when the write signal is sent and stays valid for at
least two ms.

REPRESENTATION:
The data will be using an ASCII representation.

   DATA-ITEM: SIGNALS

Signals specifying address and data

 SYNONYM: SIGS

 TYPE: RECORD

PURPOSE:
  not available

TIMING:
  not available

REPRESENTATION:
  not available
```

Activity Interface Report

An activity interface report presents the input and output for a given activity. This template produces an interface report for a given activity. No text formatter is used—formatting is done using `WRITE` statements.

To understand the report, consider the activity chart in the following figure.



The chart shows an activity named `SC_ACTIVITIES` with its subactivities. Flow-lines indicating the flow of events, conditions, and data-items enter and exit `SC_ACTIVITIES`. Some of these flow-lines lead directly to, or exit from, a particular subactivity, whereas others lead to, or exit from, the parent activity (`SC_ACTIVITIES` itself). The interface report lists the elements flowing through these flow-lines, indicating whether they are events, conditions, or data-items.

For example:

```
DATA_ITEM   ACCEL_DEFLECTION
EVENT       ACTIVATE_CRUISE
EVENT       CLOCK
DATA_ITEM    MILEAGE
```

Note that flow-lines also lead from one subactivity to another. These are not considered as inputs and outputs to the main activity of the chart, and therefore information concerning them does not appear in the interface report.

The template first searches for `a_flow_lines` using a query function. The template then extracts the conditions, events, and a data-items that flow along these lines.

act_interface Template

```
TEMPLATE act_interface;

PARAMETER
STRING activity_name;  -- the activity for which
                       --  the report is generated.

VARIABLE
   INTEGER   st;     -- return status code
   ACTIVITY  act;    -- the id of subject activity

BEGIN
   act := stm_r_ac (activity_name, st);
END;

PROCEDURE write_elements;

PARAMETER

   LIST OF ELEMENT  element_list;

VARIABLE

   ELEMENT  elm;

   INTEGER  element_type, status;

BEGIN

-- write elements
FOR elm IN element_list LOOP
  element_type := stm_r_element_type (elm, status);
   SELECT
    WHEN element_type = stm_event =>
      WRITE('\n EVENT    ',stm_r_ev_name(elm, status));
    WHEN element_type = stm_condition =>
      WRITE('\n CONDITION ',stm_r_co_name(elm, status));
    WHEN element_type = stm_data_item =>
      WRITE('\n DATA_ITEM ',stm_r_di_name(elm, status));
   END SELECT;
END LOOP;

END;


SEGMENT report;

VARIABLE
   LIST OF A_FLOW_LINE  af_list;
   LIST OF EVENT        ev_list;
   LIST OF CONDITION    co_list;
   LIST OF DATA_ITEM    di_list;
   LIST OF ELEMENT      in_list, out_list;
   ELEMENT              elm;
   INTEGER              elm_type;


BEGIN
-- write title
WRITE ('\n\n', ' ':20,'INTERFACE REPORT for ',
activity_name);
```

```
-- activity's inputs
-- input title
WRITE ('\n\n  Input elements:');
WRITE ('\n');-- get inputs

-- get input flow-lines
 af_list := stm_r_af_ext_to_target_ac({act},st);

-- get input conditions
co_list := stm_r_co_flowing_through_af(af_list,st);

-- get input events
ev_list := stm_r_ev_flowing_through_af(af_list,st);

 -- get input data_items
di_list := stm_r_di_flowing_through_af(af_list,st);

in_list:=co_list + ev_list + di_list; -- all inputs

-- order alphabetically
in_list := stm_list_sort_by_name(in_list, st);

-- write elements
write_elements(in_list);

-- activity's outputs
-- output title
WRITE ('\n\n  Output elements:');
WRITE ('\n');-- get outputs

-- get output flow-lines
af_list := stm_r_af_ext_from_source_ac ({act}, st);

-- get output conditions
co_list := stm_r_co_flowing_through_af (af_list, st);

-- get output events
ev_list := stm_r_ev_flowing_through_af (af_list, st);

-- get output data_items
di_list := stm_r_di_flowing_through_af (af_list, st);

-- all outputs
out_list := co_list + ev_list + di_list ;

-- order alphabetically
out_list := stm_list_sort_by_name (out_list, st);

-- write elements
write_elements (out_list);

END;
```

Activity Interface Report Initiation Section

Declarations and assignments in the initiation section apply throughout the template. In the initiation section, one parameter is declared, `activity_name`, which can be assigned via a special form before template execution. This is the name of the activity for which the interface report is to be generated. In addition, two variables are declared:

- `st` - The status code returned by the database extraction function
- `act` - The ID of the activity for which the report is produced

The body of the initiation section has only one statement between the `BEGIN` and `END` keywords. This is an assignment statement that assigns the activity's ID to the `act` variable. Because elements are identified in database extraction functions by their IDs and not their names, you must first find the ID for the activity before you can extract information about it. To do this, use the activity's name as an argument in a database extraction function that returns the corresponding activity's ID. By assigning this ID to the `act` variable, you can later use it throughout the template whenever you want to extract information about the activity.

Activity Interface Report Segment Section

The long template segment begins with the segment identifier line, SEGMENT report, `SEGMENT report;`. This line is used to attach a name to the segment that follows. In templates possessing more than one segment, you can use the name to select a segment for exclusive execution. Next, there is a declaration section. followed by the segment body that consists of execution statements enclosed between the `BEGIN` and `END` keywords.

Declarations

The declaration section consists of variables used to construct the list of element inputs and outputs. The first four variables—`af_list`, `ev_list`, `co_list`, and `di_list`—represent lists of a-flow-line elements and the events, conditions, and data-items that flow along them. The next two variables, `in_list` and `out_list`, represent the union of the list of elements represented by the variables `ev_list`, `co_list`, and `di_list`. The variable `in_list` represents a combined list of input elements. The variable `out_list` represents a combined list of output elements.

The next variable, `elm`, is used to extract elements from lists of elements using a `FOR/LOOP`. The last variable, `elm_type`, is an enumerated type that is declared as an integer. It represents the element type and is used in conjunction with a `SELECT/WHEN` construct to write out the type for each element in the list of inputs and outputs.

Producing the Headings

The first `WRITE` statement in the body (statements starting after the keyword `BEGIN`) produces the title of the report. This consists of 20 blank spaces (indicated by `' ':20`), followed by the words `INTERFACE REPORT for`, and the name of the activity for which the report is generated (this name is a parameter supplied at the time of template execution).

For a heading introducing the input elements, the template uses a second `WRITE` statement to insert two new lines, then writes the heading `Input Elements:`. This is followed by another new line command.

Building Element Lists

The next part of the template builds lists of input elements using database extraction functions. First, the template extracts from the database all the a-flow-line elements that have the subject activity (or its subactivities) as their target. This list is assigned to the variable `af_list`. Note that the database extraction function that extracts this list has its parameter, `act`, enclosed in braces. Braces indicate a list of items; because this particular extraction function is a query function, the parameter must consist of a list. In this case, however, the "list" is only one activity ID, represented by `act`.

The template builds three separate lists of elements:

- A list of input conditions, represented by the variable `co_list`
- A list of input events, represented by the variable `ev_list`
- A list of input data-items, represented by the variable `di_list`

Each of these variables is assigned values through a database extraction function that uses the variable `af_list` as a parameter. In other words, once you have extracted all the a-flow-line elements related to the specified activity, you extract the conditions, events, and data-items that flow through them.

Alphabetizing and Sorting the List

To alphabetize all the elements in the lists, regardless of type, use the operation for union "+" to create a combined list of all the input elements; this list is designated by the variable `in_list`. The template uses a special utility function to sort by name the elements represented by `in_list` and reassign them to `in_list`.

Writing the Input Elements

The next part of the template segment gives instructions for writing each element's type (event, condition, or data-item), followed by its name. To do this, use a `FOR/LOOP` construct to both extract an element `elm` out of the alphabetized `in_list` and perform several statements iteratively.

The first statement in the loop extracts the type of the element represented by the variable `elm`. The template then uses a `SELECT/WHEN` construct to write the element type and name. The triggers of the `WHEN` statements are constructed so when the variable `elm_type` is equal to a particular element type, the `WRITE` statement to the right of the arrow is executed.

"Element type" is referred to as an enumerated type; each of the possible values is represented by a unique number. These numbers correspond to predefined constants (`stm_event`, `stm_condition`, and `stm_data_item`). Note that the template does not use the value of the variable `elm_type` to write the element type; doing so would cause the *integer value* to be written. Instead, the template writes the string that corresponds to the integer value.

You must use a database extraction function in the `WRITE` statements to produce the elements' names because the variable `elm` that represents the individual element (as well as the list of elements represented by the variable `in_list`) consists of *element IDs*, rather than names. To produce the element's names, use the IDs as parameters in a database extraction function that returns element names.

After the `WHEN` construct is executed and the element's type and name is written, the program loops back to the beginning of the `FOR/LOOP` and the instructions are carried out for another element. This continues until all input elements are written.

The last part of the template constructs lists of output elements and writes their types and names. This part is identical for that of input elements, except that a different function is used to build the output a-flow-line elements.

The final output is shown in the next section.

Final Output for Act_Interface Report

```
INTERFACE REPORT for sc_activities

  Input elements:

DATA_ITEM   ACCEL_DEFLECTION
EVENT       ACCEL_PRESSED
EVENT       ACTIVATE_CRUISE
EVENT       BRAKE_PRESSED
EVENT       CLOCK
CONDITION   ENGINE_RUNNING
DATA_ITEM   GEAR_POSITION
CONDITION   MEASURE_SIGNAL
DATA_ITEM   MILEAGE
EVENT       RESUME_CRUISE
EVENT       SHAFT_ROTATION
DATA_ITEM   THROTTLE_POS

   Output elements:

DATA_ITEM   CURRENT_SPEED
DATA_ITEM   MILEAGE
DATA_ITEM   THROTTLE_CONTROL
```

Template for nroff

Whereas the previous templates in this section produced reports that were formatted by DGL `WRITE` statements, the templates in the following two sections contain commands specific to various formatting languages.

After a template is executed, the generated information, together with the embedded formatting commands for a particular formatting system, is passed to the output document segments. The file can then be further processed by a formatting system so the embedded formatting instructions are executed to produce the final document.

The following sections show how to use embedded formatting commands to produce the final document, which is similar to the one documented in **Overview of Documentor**. They are different in their chapter and section headers.

This section explains the template logic for nroff. Because Interleaf has a slightly different logic, especially with regards to plot generation, it is explained in a separate section. For both templates, the initiation section is the same.

Template with nroff Commands

```
TEMPLATE example;

-- Initiation Section
PARAMETER
 STRING act_name := 'SC_ACTIVITIES';  -- the activity
                  -- for which the report is written.
 STRING plot_dev:= 'HP7475';

VARIABLE
 ACTIVITY  act_id;      -- id of 'act_name'.
 INTEGER   st;          -- return status code.
 STRING    title;       -- title of plot.
 FLOAT     acty;        -- actual height of plot.

BEGIN
   act_id := stm_r_ac(act_name, st);
END;

SEGMENT seg1;
BEGIN
  INCLUDE('nroff_glob');
  WRITE('\n.bp');
  WRITE('\n.ce\nDescription of ', act_name);
  WRITE('\n.sp\n.sh 1 Overview');
  INCLUDE('sys_overview'); -- 'sys_overview' is an
                           -- include file in which
                           -- text with formatting
                           -- command is written.
END;

SEGMENT seg2;
VARIABLE
   LIST OF ACTIVITY     ac_list;

BEGIN
/@.bp
.sh 1 "SYSTEM ACTIVITIES"
.sp
.sh 2  "Activity-chart"
.sp
 This is the chart that describes the activities
 of the system:
.br
@/
```

```
-- leave 40 lines in one page to the plot
   WRITE('\n.sp 40\n');
   title:= 'Plot of ' + act_name;
-- plot of activity-chart
stm_plt(act_id,'act_plot',6.5,9.0,'F','F',
          true,999,plot_dev,stm_plt_top,title,acty);
-- second section
/@
.bp
.sh 2 "Activities Description"
.sp
 Detailed description of each activity in chart:
@/

ac_list := stm_r_ac_physical_sub_of_ac({act_id},st);
-- Property Report
stm_rpt_dictionary(ac_list,true,true,' ');
END;
```

Initiation Section (nroff)

In the initiation section of the templates, the string parameter `act_name` represents the name of the activity for which the report is generated. The value of `act_name` can be assigned at the time of template execution. Because you can also assign an initial value to the parameter, assume that `act_name` was assigned an initial value of `SC_ACTIVITIES`.

A second parameter, `plot_dev`, represents the plotter output device on which you want to produce the plot. This output device need not be identical to the one used to print the document. For example, you might want to produce the plot and text separately, merging them together later. This parameter is assigned an initial value, which you can change in a special form at the time of template execution.

Next, the template declares four variables:

- `act_id` - The ID of the activity for which the report is generated. This variable is declared of type `ACTIVITY`.
- `st` - A return status code for the database extraction functions. It is declared as an `INTEGER`.
- `title` - The string used to pass the title to the plot tool.
- `acty` - A float that is an output parameter from the plot tool, which specifies the actual height of the plot.

Lastly, in the body of the initiation section, there is one statement, an assignment for the variable `act_id`. To make the assignment, the template uses a database extraction function that takes `act_name` as a parameter and returns the corresponding ID.

Segment 1: Heading and Report Overview (nroff)

The first template segment includes global declarations and produces a report heading and overview section. This section explains how these are produced with nroff.

Including Global Declarations

The first statement of `seg1` passes the file `nroff_glob` to the output segment. This file, which is found in the databank, must be checked out to the workarea to be used by the Documentor. The file's function is to change default settings for nroff. For example, it cancels the special setting for the underscore symbol; this is done because Statemate uses underscores in element names.

Producing the Heading and Overview (nroff)

The template uses `WRITE` statements to write nroff formatting commands to the output document segments. Note that there is a `\n` before each nroff command so each command written to the output segment starts on a separate line. For example, after executing the template, the first three `WRITE` statements cause the following to appear in the output segment `seg1`:

```
.PAGE
.CENTER; Description of SC_ACTIVITIES
.SKIP
.HL 1 Overview
```

The contents of the overview section are taken from an include file, `sys_overview`. This file resides in your workarea. As written in the comments, this file can contain formatting instructions in addition to the text. The entire file is passed verbatim to the output segment following the overview header.

After the output segment has been generated, it can then be formatted by nroff so the formatting commands are interpreted and the final, formatted document produced. For example, the formatting commands in `seg1` would cause a new page to be started; the words "Description of SC_ACTIVITIES" would then be centered in the middle of the line, followed by a "skipped" line. The next formatting command would produce a header marked by the number 1. Lastly, the text contained in `sys_overview` is formatted according to the formatting instructions in this file or, if missing, according to the nroff defaults.

Segment 2: Activity-Chart Plot and Property Report (nroff)

In the second segment of the template, `seg2`, the template produces the activity's plot and generates a property report of all of its subactivities using plot and report functions. Formatting commands are introduced where appropriate. One variable (`ac_list`) is declared, which represents the list of subactivities from which the property report is constructed.

The segment's body begins with the DGL verbatim symbol, `/@`. This command indicates that everything between it and the following `@/` symbol is to be included verbatim in the output document segment. The comments do not actually appear in the generated output, but explain the purpose of each command when interpreted by nroff. The generated output is as follows:

```
.PAGE -- starts a new page.
.HL 1 SYSTEM ACTIVITIES -- produces a header.
.SKIP -- skips a line.
.HL 2  Activity-chart -- produces a subheading.
.SKIP -- skips a line.
This is the chart that describes the activities
of the system:
.BREAK -- causes hard return (new line).
```

Note

nroff commands written in a verbatim environment must start at the extreme left of the line (otherwise, the formatter does not correctly interpret them). Do not be misled into indenting these commands in the desire to produce a more readable template.

The next `WRITE` statement passes a nroff formatting command to the output segment. This command formats the page so as to provide a place for the plot that follows. A place for the plot must be made because nroff has no inherent graphics capability. The statement passes a `SKIP` command that causes the document to skip 40 lines to leave room for the activity chart plot.

Next, the template uses an `Include Plot` statement to generate a plot of the activity for which you are producing the document. Note that the second argument is the name of the file in which the output plot is contained. The file belongs to the file system outside of Statemate. If you do not give the full path name (as in the example), the file will appear in your workarea.

The ninth argument represents the plotter type. This determines the graphical language in which the plot is generated. Because this is a template parameter, you can change its initial value in a form before executing the template.

The example template produces the plot for a pen plotter, and places it in a file called `act_plot`. Following the execution of the template you must explicitly output the plot on the particular plotting device and thereafter manually merge the plot into the blank space set aside for this. For an explanation of the other parameters of the plot, see **INCLUDE**.

Now you can give instructions for the inclusion of the property report for the subactivities. First, you include another section of formatting commands to be passed verbatim to the output segments. These include instructions for a subheading, a blank line, and a sentence introducing the property report. Next you assign a value to the variable `ac_list`, which represents the subactivities of the subject activity. This is followed by an `Include Property Report` statement, which causes the property report to appear in the output segment before formatting with its nroff commands embedded in the text. The Documentor embeds nroff commands because you attached the template to the nroff formatter at the Create Template stage. Upon formatting, these commands are interpreted along with the rest of the formatting commands.

Template for Interleaf

This section describes a template with formatting commands for Interleaf. The final formatted output is the same as for the templates explained in the previous section.

The template is as follows:

```
TEMPLATE example;

-- Initiation Section
PARAMETER
   STRING act_name;  -- the activity for which the
                     -- report is written.

VARIABLE
   ACTIVITY  act_id;      -- id of 'act_name'.
   INTEGER   st;          -- return status code.

BEGIN
   act_id := stm_r_ac(act_name, st);
END;

SEGMENT seg1;
BEGIN
 INCLUDE ('interleaf_glob');
 INCLUDE ('my_glob');
 WRITE ('\n<doc_title>');
 WRITE ('\nDescription of ',act_name);
 WRITE ('\n<sect>');
 WRITE ('\n<Autonum, List8, 1, first-Yes><TAB>Overview');
 INCLUDE ('sys_overview');  -- 'sys_overview' is an
                            -- include file in which
                            -- text with formatting
                            -- commands is written.
END;

SEGMENT seg2;


VARIABLE
   LIST OF ACTIVITY     ac_list;


   STRING               title;
```

```
   FLOAT               acty;

BEGIN
/@
<my_new_page>
<sect>
<Autonum, List8, 1><TAB>System Activities
<subsect>
<Autonum, List8, 2><TAB>Activity-chart
This is the chart that describes the activities of the system:
@/

title:= 'Plot of ' + act_name;
-- plot of activity-chart.
stm_plt (act_id, ' ', 6.5, 9.0, 'F', 'F', true, 999,
        'interleaf', stm_plt_top, title, acty);

/@
<my_new_page>
<subsect>
<Autonum, List8, 2><TAB>Activities Description
Detailed description of each activity in the chart:
@/
ac_list := stm_r_ac_physical_sub_of_ac ({act_id}, st);
-- property report
stm_rpt_dictionary (ac_list, true, true,'');
END;
```

Initiation Section (Interleaf)

The initiation section is similar to that for the nroff template; see the explanation of the nroff initiation section.

Segment 1: Heading and Report Overview (Interleaf)

Because Interleaf commands have definitions that must be specified before the commands are used, the template includes the file `interleaf_glob`, which contains these definitions. Among other definitions, it contains instructions used in producing Statemate predefined reports for Interleaf. This file is found in the databank and must be checked out to your workarea to be used by the Documentor. In addition, the template includes user-defined definitions contained in the file `my_glob`.

In the first segment, you pass both files to the output segment using `INCLUDE` statements. When the segment is formatted, the formatting command definitions included in the files are interpreted.

Producing the Heading and Overview (Interleaf)

The template uses `WRITE` statements to write the Interleaf formatting commands to the output document segments. Note that there is a `\n` before each Interleaf command so each command written to the output segment starts on a separate line.

The Interleaf commands written to the output segment and used for the heading and overview section are as follows:

```
<doc_title> -- starts a new page and defines the
               title font and centers the title.
               The definition of this command is
               included in ”my_glob”.
<sect> -- starts a new section. It defines the
          font of a section title, its margins etc.
          The definition of this command is
          included in ”my_glob”.
<Autonum, List8, 1, first-Yes> -- writes the first
                                  section number.
```

The following text appears in the output segment immediately following the contents of the include file:

```
<doc_title>
Description of SC_ACTIVITIES
<sect>
<Autonum, List8, 1, first-Yes><TAB>Overview
```

This is followed by the text of the `sys_overview` file. This is an include file and can contain text and Interleaf formatting commands.

Segment 2: Activity-Chart Plot and Property Report (Interleaf)

The second segment of the template, `seg2`, produces the activity's plot and generates a property report for all of its subactivities using plot and report functions. Formatting commands are introduced where appropriate. One variable (`ac_list`) is declared, which represents the list of subactivities from which the property is constructed.

The segment's body begins with the DGL verbatim symbol, `/@`. This command indicates that everything between it and the following `@/` symbol is to be included verbatim in the output document segment. The comments do not actually appear in the generated output, but explain the function of each command when interpreted by Interleaf. The generated output is as follows:

```
<my_new_page> -- starts a new page. The definition
                 of this command is included
                 in ”my_glob”.
<sect> -- starts a section. (see Segment 1).
<Autonum, List8, 2><TAB>System Activities -- writes
                                  --  section header.
<subsect> -- starts a subsection.
<Autonum, List8, 2><TAB> Activity-chart -- writes
                            -- subsection header.

This is the chart that describes the activities
of the system.
```

Note

Certain Interleaf commands, such as `<sect>` and `<subsect>`, that are written in a verbatim environment must start at the extreme left of the line. Otherwise, the formatter does not correctly interpret them). Do not be misled into indenting these commands in the desire to produce a more readable template.

In the case of the activity chart plot, you follow a slightly different formatting logic from that of nroff because Interleaf has a graphics capacity. Therefore, you do not have to leave space for the plot—the language automatically prepares the page for this. To include the plot in the output file, you supply an empty string as the second parameter in the `Include Plot` statement. This causes the plot information to be passed directly to the output segment. Note that the ninth parameter is `'interleaf'`, which specifies the output language in which the plot is generated. Upon formatting, Interleaf interprets the plot graphics and draws it into the document.

For an explanation of the other parameters of this plot, see the **stm_plot** function.

Now you are ready to include the property report for the subactivities. First, you include another section of formatting commands to be passed verbatim to the output segments. These include instructions for a new page, a subheading, and a sentence introducing the property report. Next, you assign a value to the variable `ac_list`, which represents the subactivities of the subject activity. This is followed by an `Include Property Report` statement. This statement causes the property report to appear in the output segment before formatting, with its Interleaf commands embedded in the text. The Documentor embeds Interleaf commands because you attached the template to the Interleaf formatter at the Create Template stage. Upon formatting, these commands are interpreted along with the rest of the formatting commands. The final output is very similar to the document in **Overview of Documentor** (but there are differences in the appearance of the headings).

Single-Element Functions

This section documents the single-element extraction functions. For each function, the following information is provided:

- Return value type
- The elements for which it is relevant
- Description
- Syntax
- Arguments
- Status codes

The two characters *xx* in the function names denote element type abbreviations. See **Element Type Abbreviations** for the list of element abbreviations.

Single-element functions provide information about discrete Statemate elements in the database.

Using single-element functions, you can retrieve any information attached to a particular element. This information is usually entered into the database via forms. Data extraction is a multi-stage procedure. Generally, when working with a Statemate element, you know the element's name (path name). You might want to know more about such an element, such as the element's synonym or what attributes are defined in the element's form. Such information can be retrieved using the single-element functions.

The retrieval process is as follows:

1. Specify the element name or synonym. Receive the element `ID`.
2. Specify the `ID` and the information requested. Receive the extracted information
3. Use the extracted information.

The element ID is an internal representation that Statemate uses to identify each element. You do not see the ID; you extract it from the database using one function and pass it along to another function that processes your information request.

The vertical ellipses in the diagram indicate that multiple functions can be called in succession for the same element. For example, you can call a function to return an element's synonym, then immediately call a different function to return the contents of the same element's description field.

Calling Single-Element Functions

As shown in the diagram, extracting information from your database is at least a two-stage process.

- **Stage 1—**Pass the element name or synonym as a function argument to get the element ID. The function calling sequence is as follows:

  `stm_r_`*`xx`* `(name, status)`

  In this syntax:

  - **stm_r_**—Designates the function as a Statemate database retrieval function.
  - **xx**—The two-character element type abbreviation.
  - **name**—The name of the element for which information is requested. The input argument name contains the name (path name) or synonym that uniquely identifies the element of interest. The name can be a variable or a literal string (enclosed by single apostrophe marks).
  - **status**—The return function status code.

  For example:

  ```
  stm_r_st('S1',status)
  ```

  This function call returns the ID for state `S1`. The value returned by the function is a Statemate element of the type specified by *xx*. In this example, the value returned by the function is of type `STATE`.

- **Stage 2**—Pass the element ID as a function argument to get the information requested. The function calling sequence is as follows:

  `stm_r_`*`xx`*`_info (inarg, ..., status)`

  or

  `stm_r_info (inarg, ..., status)`

  In this syntax:

  - **stm_r_**—Designates the function as a Statemate database retrieval function.
  - **xx**—The two-character element type abbreviation. Note that in some functions, these two characters are omitted.
  - **info**—The type of information to be extracted from the database.
  - **inarg**—The required input arguments.
  - **status**—The return function status code.

For example:

```
stm_r_ac_description (a, status)
or
stm_r_description (a, status)
```

This function call retrieves the contents of the **Description** field for the activity whose ID is contained in the variable `a`.

There is one function whose calling sequence differs from that shown above. This function, `stm_r_element_type`, receives an element ID as input and returns the element type. The function returns an enumerated type value of the form "`stm_state`", "`stm_activity`", and so on.

Single-Element Function Input Arguments

The following table lists the input arguments for single-element functions.

Argument	Function	DGL Data Type
name	The name of the Statemate element. It can be an element name, path name, or synonym, including the chart name, (for example, K:L.M).	String
element ID	The value that Statemate uses to identify each element in the database. Statemate assigns a unique ID to every element.	Statemate element (`ELEMENT`, `STATE`, `EVENT`, and so on). For example, if you call a function that operates on events, the element ID must be defined to be of type `EVENT` (or `ELEMENT`).
attribute name	Name of an attribute defined in the form for a Statemate element (in the attribute field).	String
begin keyword	The string of text appearing in the long description attached to the specified Statemate element. This string represents the beginning of the portion of the long description that you want to extract from the database.	String
end keyword	The string of text appearing in the long description attached to the specified element. This string represents the end of the portion of the long description that you want to extract from the database.	String
filename	The path name of a system file. Long descriptions (and portions thereof) are copied to the system file specified in this argument. If this is an empty string (`''` - two contiguous apostrophes), the Documentor creates a temporary file where it stores the text. The name of this temporary file is returned by the function.	String

Examples of Single-Element Function Calls

This section shows how to use single-element function calls to perform common tasks.

Single-Element Function Example 1

Suppose you want to find the synonym and the short description for a state `S1`, as they appear in the state's form. Assume that the path name `SSS.S1` uniquely identifies the state. Include the following code in your Documentor template:

```
VARIABLE
   STATE      state_id;
   STRING     state_desc, state_syn;
   INTEGER    status;
              .
              .
state_id  := stm_r_st ('SSS.S1', status);
state_syn := stm_r_st_synonym (state_id, status);
state_desc:= stm_r_st_description (state_id, status);
              .
              .
```

This example uses two consecutive function calls to extract first the synonym, then the short description of the same element. The assigned variable and the function return value must have compatible data types. Therefore, `state_id` is declared as `STATE`. (`state_id` could also be declared as `ELEMENT`.)

Single-Element Function Example 2

The following example shows functions that return enumerated type values.

```
VARIABLE
   STATE      state_id;
   STRING     state_name, state_type;
   INTEGER    st_type;
   INTEGER    status;
              .
              .
state_id := stm_r_st (state_name, status);
st_type := stm_r_st_type (state_id, status);
   SELECT
      WHEN st_type = stm_st_or =>
         state_type := 'or';
         WHEN st_type = stm_st_and =>
         state_type := 'and';
              .
              .
   END SELECT;
WRITE ('The state ',state_name, 'is of type ',
   state_type);
```

This example queries the database to determine the type of the state in `state_name`. When the type is determined, the name and type of the state are printed out.

Single-Element Function Example 3

The following example shows how to include a portion of the long description of the state `S1` in a document. The section extracted is the text appearing between the strings `'!BEGIN'` and `'!END'`.

```
VARIABLE
   STATE     state_id;
   STRING    descr_file;
             .
             .
state_id := stm_r_st ('S1', status);
descr_file := stm_r_st_keyword (state_id, '!BEGIN',
   '!END','',status);
INCLUDE (descr_file);
             .
             .
             .
```

The fourth input parameter (the empty string) of the function `stm_r_st_keyword` determines the name of the file to which the extracted text is written. If the string is empty, as in this case, the Documentor creates a temporary file where it stores the text. The name of this temporary file is returned by the function. Using the `INCLUDE` statement, you write the text to the generated document.

List of Functions

As previously mentioned, the extraction functions take the form `stm_r_<element_type><task>`. For example, `stm_r_uc_attr_name` returns the names of attributes associated with the specified use case. Because this function can retrieve values for other elements besides use cases, it is denoted as `stm_r_xx_attr_name`. This function would be included in the A section (for `attr_name`).

The functions are as follows:

Function	Description
stm_r_sb_action_lang	Returns the action language of the specified subroutine.
stm_r_sb_action_lang_expression	Returns the action language expression of the specified subroutine.
stm_r_sb_action_lang_local_data	Returns the action language local data associated with the specified subroutine.
stm_r_actual_parameter_exp	Returns the actual binding of the formal parameter name in the specified instance chart or component.
stm_r_actual_parameter_type	Returns the type of the formal parameter name in the specified instance chart or component.
stm_r_elem_in_ddb_list	Searches for the specified element in the list created by the properties browser.
stm_r_sb_ada_user_code	Returns the Ada code that was manually written by the user for the specified subroutine.

stm_r_sb_ansi_c_user_code	Returns the ANSI C code that was manually written by the user for the specified subroutine.
stm_r_xx	Retrieves the element ID of the specified element.
stm_r_xx_array_lindex	Returns the left index of an element array.
stm_r_xx_bit_array_rindex	Returns the right index of an element array.
stm_r_xx_attr_enforced	Returns the enforced attributes specified by `attr_name`.
stm_r_xx_attr_name	Returns the names of attributes associated with the specified element.
stm_r_xx_attr_val	Returns the attribute values associated with a particular attribute name for the specified element.
stm_r_xx_bit_array_lindex	Returns the left index of a bit array.
stm_r_xx_bit_array_rindex	Returns the right index of a bit array.
stm_r_xx_cbk_binding	Retrieves the callback binding for the specified element.
stm_r_xx_cbk_binding_enable	Retrieves the enabled callback bindings for the specified element.
stm_r_xx_cbk_binding_expression	Retrieves the callback binding expression for the specified element.
stm_r_xx_cbk_binding_expression_hyper	Retrieves the callback binding expressions.
stm_r_tt_cell	Returns the contents of the specified cell in the given truth table.
stm_r_tt_cell_type	Returns the data-type of the specified cell in the given truth table.
stm_r_changes_log	Documents all the changes made to the specified charts in a log file.
stm_r_xx_chart	Returns the chart ID for the specified element.
stm_r_xx_combinationals	Returns a list of strings. Each element of this list holds one combinational assignment, which is connected to the specified element.
stm_r_sb_connected_chart	Returns the ID of the procedural statechart connected to the specified subroutine.
stm_r_xx_containing_fields	Returns a list of union or record elements that contain fields.
stm_r_ch_creation_date	Returns the date (as a string) on which the specified chart was created.
stm_r_ch_creator	Returns the name of the Statemate user who created the specified chart.
stm_r_xx_data_type	Returns the element subtype, including its data type and data structure.
stm_r_xx_definition_type	Returns the definition type of the specified textual element.
stm_r_xx_desc_file	Returns the description file for the specified element.
stm_r_xx_description	Returns the short description of the specified element.
stm_r_design_attr	Retrieves the design attributes of an element in a form of a list of strings.
stm_r_xx_displayed_name	Returns the name of the chart as it appears in the graphic editor where the specified element is located.
stm_r_ddb_list_names	Returns the names of the lists created by the properties browser.

stm_r_element_type	Returns the element type of the specified element.
stm_r_xx_expr_hyper	Returns the definition expression of the specified element found in the **Definition** field of the element's form, including hyperlinks to referenced elements.
stm_r_xx_expression	Returns the definition expression of the specified element found in the **Definition** field of the element's form.
stm_r_xx_ext_link	Retrieves the external file link for the specified element.
stm_r_uc_ext_point_def	Retrieves the extension point definitions for the specified use case diagram.
stm_r_formal_parameter_names	Retrieves the list of names of formal parameters that appear in the bindings of instance boxes and components.
stm_r_global_interface_report	Creates a global interface report from the specified input list.
stm_r_sb_global_data	Returns the global data associated with the specified subroutine.
stm_r_hyper_key	Retrieves the unique key for the specified element.
stm_r_md_implementation	Retrieves the implementation type for the specified module.
stm_r_included_gds	Returns the list of global definition sets contained in the specified chart.
stm_r_msg_included_in_ord_insig	Returns a list of messages that are bounded by an order-insignificant element.
stm_r_cd_info	Retrieves the description of the specified continuous chart.
stm_r_inherited_gds	Retrieves the list of global definition sets that are "inherited" (included indirectly) by the specified chart.
stm_r_xx_instance_name	Retrieves the name of an instance as it appears in the chart for the specified hierarchical Statemate element.
stm_r_xx_keyword	Returns a portion of the element's long description.
stm_r_sb_kr_c_user_code	Returns the K&R C code that was manually written by the user for the specified subroutine.
stm_r_xx_labels	Returns a list of strings that consists of all the labels of the specified compound transition.
stm_r_xx_labels_hyper	Returns a list of strings of message or transition labels, with hyperlinks to referenced elements.
stm_r_local_interface_report	Creates a local interface report from the specified input list.
stm_r_xx_longdes	Returns the long description attached to the specified element.
stm_r_xx_max_val	Returns the maximum value of the specified element.
stm_r_xx_min_val	Returns the minimum value of the specified element.
stm_r_xx_mini_spec	Returns a string with mini-spec reactions or actions.
stm_r_ac_mini_spec_hyper	Returns a string with the mini-spec, including hyperlinks to referenced elements.
stm_r_xx_mode	Returns the parameter or router mode.
stm_r_ch_modification_date	Returns the date on which the version of the chart in the workarea was saved to the database.
stm_r_xx_name	Returns the element name.

stm_r_next_msg	Returns the message after (in time) the decomposed sequence diagram.
stm_r_xx_note	Returns the notes in the specified element.
stm_r_tt_num_of_col	Retrieves the number of columns in the specified truth table.
stm_r_tt_num_of_in	Retrieves the number of input columns in the specified truth table.
stm_r_tt_num_of_out	Retrieves the number of output columns in the specified truth table.
stm_r_tt_num_of_row	Retrieves the number of rows in the specified truth table.
stm_r_uc_num_of_scen	Retrieves the number of scenarios for the specified use case.
stm_r_xx_number_of_bits	Returns the number of bits in the specified element.
stm_r_xx_of_enum_type	Returns the enumeration type ID (a user-defined type) for the specified element.
stm_r_xx_of_enum_type_name_type	Returns the enumerated name and type for the specified element.
stm_r_ord_insig_defined_in_ch	Returns the origin of the specified requirement record.
stm_r_parameter_binding	Returns the parameter expression from the generic charts and components.
stm_r_parameter_mode	Retrieves the parameter mode, including subroutine parameters and the parameters of generic charts and components.
stm_r_sb_parameters	Returns the parameters of the specified subroutine.
stm_r_en_parent	Returns the local data of the procedural statechart implemented by the specified subroutine.
stm_r_previous_msg	Returns the message previous (in time) to the decomposed sequence diagram.
stm_r_sb_proc_sch_local_data	Retrieves the local data of the procedural statechart implemented by the specified subroutine.
stm_r_md_purpose	Retrieves the purpose of the module.
stm_r_xx_reactions	Returns the static reactions of the specified state.
stm_r_sb_return_type	Retrieves the subroutine's return type.
stm_r_sb_return_user_type	Returns the user-defined type ID returned by the specified subroutine.
stm_r_sb_return_user_type_name	Retrieves the subroutine's return user type and name type.
stm_r_tt_row	Retrieves the values in the specified row in the truth table.
stm_r_uc_scen	Retrieves the scenario, based on the specified index number.
stm_r_uc_scen_attr_name	Returns the names of attributes associated with the specified use case scenario.
stm_r_uc_scen_attr_val	Retrieves attribute values associated with a particular attribute name for the specified use case scenario.
stm_r_sd_scope	Retrieves the scope of the specified sequence diagram.
stm_r_xx_select_implementation	Returns the implementation type of the specified element.
stm_r_st_static_reactions	Returns the static reactions defined for the specified state element.

stm_r_st_static_reactions_hyper	Returns a string with the static reactions, including hyperlinks to referenced elements.
stm_r_xx_string_length	Returns the string length of the specified element.
stm_r_xx_structure_type	Returns the structure or type of the specified textual element.
stm_r_ac_subroutine_bind	Returns the subroutine binding connected to the specified activity.
stm_r_ac_subroutine_bind_enable	Determines whether the subroutine bound to the specified activity is enabled or disabled.
stm_r_ac_subroutine_bind_expr	Returns the subroutine binding expression connected to the specified activity.
stm_r_xx_synonym	Returns the synonym of the specified element.
stm_r_ac_termination	Returns the activity termination type specified in the activity's form.
stm_r_xx_truth_table	Returns the elements that are implemented as truth tables.
stm_r_xx_truth_table_expression	Returns the truth table expression for all the named elements.
stm_r_sb_truth_table_local_data	Returns a list of local data elements defined in truth tables that are related to the input subroutine.
stm_r_xx_type	Returns the element subtypes for the specified element.
stm_r_xx_type_expression	Returns the type expression for the specified element.
stm_r_xx_uniquename	Returns the unique path name for the specified element.
stm_r_ch_usage_type	Returns the usage type for the specified chart.
stm_r_xx_user_type	Returns the user-defined type ID referenced by the specified element.
stm_r_xx_user_type_name_type	Returns the name type of the user-defined type referenced by the specified element.
stm_r_ch_version	Returns the version of the specified chart.
stm_r_gds_visibility_mode	Returns the visibility mode for the specified global definition set (GDS).
stm_r_msg_where_tc_begins	Returns the message where the timing constraint begins.
stm_r_msg_where_tc_ends	Returns the message where the timing constraint ends.
stm_r_xx_chart	Returns the chart ID for the specified element.
stm_r_xx_longdes	Retrieves the long description attached to the specified element.
stm_r_xx_attr_name	Returns the names of attributes associated with the specified element. Attributes are associated with elements via element forms.
stm_r_xx_attr_val	Retrieves attribute values associated with a particular attribute name for the specified element.
stm_r_sb_connected_statechart	Returns the enforced attributes specified by attr_name.
stm_r_sb_connected_statechart	Returns the ID of the procedural Statechart connected to the specified subroutine.
stm_r_sb_connected_flowchart	Returns the ID of the Flowchart connected to the specified subroutine.
stm_r_sb_proc_fch_local_data	Retrieves the local data of the Flowchart implemented by the specified subroutine.

stm_r_xx_des_attr_val	Retrieves attribute values associated with a particular attribute name for the specified element.
stm_r_xx_des_attr_name	Returns the names of attributes associated with the specified element. Attributes are associated with elements via element forms.
stm_r_tt_cell_hyper	Retrieves the contents of the specified cell in the given truth table, including hyperlinks to referenced elements.
stm_r_xx_default_val	Returns a list of strings that represents a row in the truth table, including hyperlinks to referenced elements. Each string in the list includes the text in the truth table cell. The row's index range is [0..num_of_rows-1]. Row 0 returns the list of table header strings.
stm_r_xx_default_val	Returns the default value associated with the specified element.
stm_r_component_param_binding	Returns the value bound to the formal parameter of the specified component's instance
stm_r_component_param_mode	Returns the mode of the formal parameter of the specified component's instance.
stm_r_stubs_names	Returns a list of stub names of the specified component instance.
stm_r_information_stub_names	Returns a list of stub names flowing through an info-flow of the specified component instance.

stm_r_xx

Retrieves the element ID of the specified element. This ID is an internal representation that Statemate uses to identify each element in the database. Because Statemate requires the ID to locate elements, this function is very often the first one called when using database exttraction functions.

Function type

Statemate element

For elements

action	an
activity	ac
actor	actor
boundary box	bb
chart	ch
condition	co
data-item	di
data-store	ds
enumerated value	en
event	ev
field	fd
function	fn
information-flow	if
lifeline	ll
module	md
router	router
state	st
subroutine	sb
subroutine parameter	sp
use case	uc
user_defined_type	dt

Syntax

```
stm_r_xx (name, status)
```

Arguments

Argument	Input/ Output	Type	Description
name	In	STRING	A Statemate element name or synonym. Note the following: • This can be an element name (path name) or synonym. Hierarchical elements must be identified uniquely by specifying a unique path name. • The name can include the chart name (for example, A:B). • The name is not case-sensitive.
status	Out	INTEGER	The function status code. The data type of the return value must be declared to be either a Statemate element (STATE, EVENT, and so on) or ELEMENT.

Status Codes

- stm_success
- stm_illegal_address
- stm_illegal_name
- stm_name_not_found
- stm_name_not_unique

Example

Identify the ID of an event EV1. Once the ID has been determined, you can use it to retrieve information about EV1 from the database, as follows:

```
VARIABLE
      EVENT      ev_id;
      INTEGER    status;
      STRING     synonym;
                    .
                    .
                    .
ev_id := stm_r_ev ('EV1', status);
IF status = stm_success
 THEN
  synonym:= stm_r_ev_synonym (ev_id, status);
                    .
                    .
                    .
```

The ID for EV1 is assigned to the variable ev_id. Note that ev_id is declared to be of type EVENT.

stm_r_sb_action_lang

Retrieves the action language of the specified subroutine.

Function type

```
LIST OF STRING
```

For elements

subroutine	sb

Syntax

```
stm_r_sb_action_lang (sb_id, status)
```

Arguments

Argument	Input/Output	Type	Description
sb_id	In	Statemate element	The element ID
status	Out	INTEGER	The function status code

Status Codes

- stm_success
- stm_id_out_of_range
- stm_id_not_found
- stm_missing_statemate_action_lang
- stm_unresolved

stm_r_sb_action_lang_expression

Retrieves the action language expression of the specified subroutine.

Function type

`STRING`

For elements

subroutine	sb

Syntax

```
stm_r_sb_action_lang_expression (sb_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`sb_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_missing_statemate_action_lang`
- `stm_unresolved`

stm_r_sb_action_lang_local_data

Retrieves the action language local data associated with the specified subroutine.

Function type

```
LIST OF STATEMATE ELEMENTS
```

For elements

subroutine	sb

Syntax

```
stm_r_sb_action_lang_local_data (sb_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`sb_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_missing_local_data`
- `stm_id_not_found`
- `stm_id_out_of_range`

stm_r_actual_parameter_exp

Returns the actual binding of the formal parameter name in the specified instance chart or component.

Function type

```
INTEGER
```

For elements

activity	ac
condition	co
data-item	di
event	ev

Syntax

```
stm_r_actual_parameter_exp (xx_inst_boxid, formal_param_name, status)
```

Arguments

Argument	Input/ Output	Type	Description
`inst_boxid`	In	Statemate element	The element ID.
`formal_param_name`	In	`STRING`	The formal parameter name. If this is a data-element (from the information stub matrix in the DDE), the function returns the corresponding data-element. If this argument is the stub's name, the function returns the information flowing on the arrow connected to that stub.
`status`	Out	`int`	The function status code.

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_name_not_found`

stm_r_actual_parameter_type

Returns the type of the formal parameter name in the specified instance chart or component.

Note: If there is an information-flow stub, the function returns `stm_information_flow`.

Function type

`STRING`

For elements

activity	ac
condition	co
data-item	di
event	ev

Syntax

```
stm_r_actual_parameter_type (inst_boxid, formal_param_name, status)
```

Arguments

Argument	Input/ Output	Type	Description
`inst_boxid`	In	Statemate element	The element ID
`formal_param_name`	In	`STRING`	The formal parameter name
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_name_not_found`

stm_r_elem_in_ddb_list

Searches for the specified element in the list created by the properties browser.

Function type

```
LIST OF ELEMENT stm_list
```

Syntax

```
stm_r_elem_in_ddb_list (list_name, &status)
```

Arguments

Argument	Input/Output	Type	Description
`list_name`	In	`char * STRING`	The input list of elements
`status`	Out	`int INTEGER`	Function status code

Status Codes

- `stm_no_such_list`
- `stm_success`

stm_r_sb_ada_user_code

Returns the Ada code that was manually written for the specified subroutine.

Function Type

```
LIST OF STRING
```

For elements

subroutine	sb

Syntax

```
stm_r_ada_user_code (sb_id, status)
```

Arguments

Argument	Input/Output	Type	Description
sb_id	In	Statemate element	The element ID
status	Out	INTEGER	The function status code

Status Codes

- stm_success
- stm_id_out_of_range
- stm_missing_user_code

stm_r_sb_ansi_c_user_code

Returns the ANSI C code that was manually written for the specified subroutine.

Function type

```
LIST OF STRING
```

For elements

subroutine	sb

Syntax

```
stm_r_sb_ansi_c_user_code (sb_id, status)
```

Arguments

Argument	Input/Output	Type	Description
sb_id	In	Statemate element	The element ID
status	Out	INTEGER	The function status code

Status Codes

- stm_success
- stm_id_out_of_range
- stm_missing_user_code

stm_r_st_combinationals

Returns a list of strings. Each element of the list holds one combinational assignment, which is connected to the specified element.

You can call this function without indicating the specific element type, as follows:

```
stm_r_combinationals (id, status)
```

Function type

```
LIST OF STRING
```

For elements

activity	ac
chart	ch

Syntax

```
stm_r_xx_combinationals (xx_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`xx_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_error_in_file`
- `stm_missing_field`
- `stm_missing_label`
- `stm_missing_name`
- `stm_file_not_found`
- `stm_id_not_found`
- `stm_id_out_of_range`
- `stm_illegal_parameter`

stm_r_xx_array_lindex

Returns the left index of an element array.

You can call this function without indicating the specific element type, as follows:

```
stm_r_array_lindex (id, status)
```

Function type

`STRING`

For elements

condition	co
data-item	di
event	ev
field	fd
local data	ld
subroutine parameter	sp
user-defined type	dt

Syntax

```
stm_r_xx_array_lindex (xx_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`xx_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_not_found`
- `stm_id_out_of_range`

stm_r_xx_array_rindex

Returns the right index of an element array.

You can call this function without indicating the specific element type, as follows:

```
stm_r_array_rindex (id, status)
```

Function type

`STRING`

For elements

condition	co
data-item	di
event	ev
field	fd
local data	ld
subroutine parameter	sp
user-defined type	dt

Syntax

```
stm_r_xx_array_rindex (xx_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`xx_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_not_found`
- `stm_id_out_of_range`

stm_r_xx_attr_enforced

Returns the enforced attributes specified by `attr_name`.

You can call this function without indicating the specific type, as follows:

```
stm_r_attr_enforced (id, attr_name, attr_val, status)
```

Function type

BOOLEAN

For elements

action	an
activity	ac
actor	actor
boundary box	bb
chart	ch
condition	co
data-item	di
data-store	ds
event	ev
field	fd
information-flow	if
lifeline	ll
module	md
router	router
state	st
subroutine	sb
use case	uc
user-defined type	dt

Syntax

```
stm_r_xx_attr_enforced (xx_id, attr_name, attr_val, status)
```

Arguments

Argument	Input/Output	Type	Description
`xx_id`	In	Statemate element	The element ID.
`attr_name`	In	`STRING`	The attribute name.
`attr_val`	In	`STRING`	The attribute value.
`status`	Out	`INTEGER`	The function status code. If no attributes exist for the specified element, `status` receives the value `stm_attribute_name_not_found`.

Status Codes

- `stm_success`
- `stm_attribute_name_not_found`
- `stm_id_not_found`
- `stm_id_out_of_range`
- `stm_illegal_name`
- `stm_unresolved`

stm_r_xx_attr_name

Returns the names of attributes associated with the specified element. Attributes are associated with elements via element forms.

You can call this function without indicating the specific element type, as follows:

```
stm_r_attr_name (id, status)
```

Function type

```
LIST OF STRING
```

For elements

action	an
activity	ac
actor	actor
boundary box	bb
chart	ch
condition	co
data-item	di
data-store	ds
event	ev
field	fd
information-flow	if
lifeline	ll
module	md
router	router
state	st
subroutine	sb
use case	uc
user-defined type	dt

Syntax

```
stm_r_xx_attr_name (xx_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`xx_id`	In	Statemate element	The element ID.
`status`	Out	`INTEGER`	The function status code. If no attributes exist for the specified element, `status` receives the value `stm_attribute_name_not_found`.

Status Codes

- `stm_success`
- `stm_attribute_name_not_found`
- `stm_id_not_found`
- `stm_id_out_of_range`
- `stm_unresolved`

Example

To perform operations on the attributes of the state `WAIT`, retrieve a list of `WAIT`'s attribute names. The template contains the following statements:

```
VARIABLE
    STATE              st_id;
    LIST OF STRING     attr_list;
    STRING             attrib;
    INTEGER            status;
                    .
                    .
                    .
st_id := stm_r_st ('WAIT', status);
attr_list := stm_r_st_attr_name (st_id, status);
FOR attrib IN attr_list LOOP
                    .
                    .
                    .
```

`attr_list` contains a list of attribute names for `WAIT`. In the `FOR` loop, perform the operations on each item in the list of attributes (such as retrieving and printing the corresponding values).

stm_r_xx_attr_val

Retrieves attribute values associated with a particular attribute name for the specified element.

You can call this function without indicating the specific element type, as follows:

```
stm_r_attr_val (id, attr_name, status)
```

Function type

```
LIST OF STRING
```

For elements

action	an
activity	ac
actor	actor
boundary box	bb
chart	ch
condition	co
data-item	di
data-store	ds
event	ev
field	fd
information-flow	if
lifeline	ll
module	md
router	router
state	st
subroutine	sb
use case	uc
user-defined type	dt

Syntax

```
stm_r_xx_attr_val (xx_id, attr_name, status)
```

Arguments

Argument	Input/Output	Type	Description
`xx_id`	In	Statemate element	The element ID.
`attr_name`	In	`STRING`	The attribute name. The attribute name is not case-sensitive.
`status`	Out	`INTEGER`	The function status code. If `attr_name` does not exist for the specified element, `status` receives the value `stm_attribute_name_not_found`.

Note the following:

- Attribute values might exist for attributes with no name. Therefore, if you supply contiguous apostrophes (`''`) for `attr_name`, you retrieve all values for unnamed attributes.
- In most cases, attributes have only one value. However, there are some cases where more than one attribute value is simultaneously meaningful. For example, a module has an attribute `implementation`. The attributes `software` and `hardware` might both be meaningful for some modules. Therefore, Statemate provides the capability of assigning multiple values to attributes, and the function returns a list of these values. When there is a single value, the list consists of one component.

Status Codes

- `stm_success`
- `stm_attribute_name_not_found`
- `stm_id_not_found`
- `stm_id_out_of_range`
- `stm_illegal_name`
- `stm_unresolved`

Example

The following code prints the attribute values of all the attributes defined for the state `WAIT`:

```
VARIABLE
    STATE              st_id;
    LIST OF STRING     attr_list;
    STRING             attrib, value;
    INTEGER            status;
                    .
                    .
                    .
st_id := stm_r_st ('WAIT', status);
attr_list :=stm_r_st_attr_name (st_id, status);
FOR attrib IN attr_list LOOP
   FOR value IN stm_r_st_attr_val (st_id, attrib, status)
       LOOP
          WRITE ('\n', attrib,' is ',value);
       END LOOP;
   END LOOP;
                    .
                    .
                    .
```

`attr_list` contains a list of attributes for `WAIT`. Write the attribute values for each item in this list to the document.

stm_r_xx_bit_array_lindex

Returns the left index of a bit array.

You can call this function without indicating the specific element type, as follows:

```
stm_r_bit_array_lindex (id, status)
```

Function type

```
STRING
```

For elements

data-item	di
field	fd
local data	ld
subroutine parameter	sp
user-defined type	dt

Syntax

```
stm_r_xx_bit_array_lindex (xx_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`xx_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_not_found`
- `stm_id_out_of_range`

stm_r_xx_bit_array_rindex

Returns the right index of a bit array.

You can call this function without indicating the specific element type, as follows:

```
stm_r_bit_array_rindex (id, status)
```

Function type

`STRING`

For elements

data-item	di
field	fd
local data	ld
subroutine parameter	sp
user-defined type	dt

Syntax

```
stm_r_xx_bit_array_rindex (xx_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`xx_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_not_found`
- `stm_id_out_of_range`

stm_r_xx_cbk_binding

Retrieves the callback binding for specified elements.

You can call this function without indicating the specific element type, as follows:

```
stm_r_cbk_binding (id, status)
```

Function type

```
LIST OF STRING
```

For elements

activity	ac
condition	co
data-item	di
event	ev
state	st

Syntax

```
stm_r_xx_cbk_binding (xx_id, status)
```

Arguments

Argument	Input/Output	Type	Description
xx_id	In	Statemate element	The element ID
status	Out	INTEGER	The function status code

Status Codes

- stm_success
- stm_id_not_found
- stm_id_out_of_range
- stm_missing_cbk_binding
- stm_unresolved

stm_r_xx_cbk_binding_enable

Retrieves the enabled callback bindings.

You can call this function without indicating the specific element type, as follows:

```
stm_r_cbk_binding_enable (id, status)
```

Function type

```
LIST OF STATEMATE ELEMENTS (predefined constant)
```

For elements

activity	ac
condition	co
data-item	di
event	ev
state	st

Syntax

```
stm_r_xx_cbk_binding_enable (id, status)
```

Arguments

Argument	Input/Output	Type	Description
`xx_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_not_found`
- `stm_id_out_of_range`
- `stm_missing_cbk_binding`

Return Values

Although the return value of this function is of type `INTEGER`, the Documentor enables you to reference this value by name. The name is defined internally as a predefined constant in DGL.

Element	Element Subtype
`activity`	`stm_ac_cbk_enable`
	`stm_ac_cbk_disable`
	`stm_ac_cbk_bind_missing`
`condition`	`stm_co_cbk_enable`
	`stm_co_cbk_disable`
	`stm_ac_cbk_bind_missing`
`data-item`	`stm_di_cbk_enable`
	`stm_di_cbk_disable`
	`stm_di_cbk_bind_missing`
`event`	`stm_ev_cbk_enable`
	`stm_ev_cbk_disable`
	`stm_ev_cbk_bind_missing`
`state`	`stm_st_cbk_enable`
	`stm_st_cbk_disable`
	`stm_st_cbk_bind_missing`

stm_r_xx_cbk_binding_expression

Retrieves the callback binding expressions.

You can call this function without indicating the specific element type, as follows:

```
stm_r_cbk_binding_expression (id, status)
```

Function type

`STRING`

For elements

activity	ac
condition	co
data-item	di
event	ev
state	st

Syntax

```
stm_r_xx_cbk_binding_expression (id, status)
```

Arguments

Argument	Input/Output	Type	Description
`xx_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_not_found`
- `stm_id_out_of_range`
- `stm_missing_cbk_binding`
- `stm_unresolved`

stm_r_xx_cbk_binding_expression_hyper

Retrieves the callback binding expressions.

You can call this function without indicating the specific element type, as follows:

```
stm_r_cbk_binding_expression_hyper (id, status)
```

Function type

`STRING`

For elements

activity	ac
condition	co
data-item	di
event	ev
state	st

Syntax

```
stm_r_xx_cbk_binding_expression_hyper (id, status)
```

Arguments

Argument	Input/Output	Type	Description
`xx_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_not_found`
- `stm_id_out_of_range`
- `stm_missing_cbk_binding`
- `stm_unresolved`

stm_r_tt_cell

Retrieves the contents of the specified cell in the given truth table.

Function type

`STRING`

For elements

action	an
activity	ac
subroutine	sb

Syntax

```
stm_r_tt_cell (el, row_num, col_num, status)
```

Arguments

Argument	Input/Output	Type	Description
`el`	In	Statemate element	The element ID
`row_num`	In	`INTEGER`	The row number of the cell
`col_num`	In	`INTEGER`	The column number of the cell
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_id_not_found`
- `stm_unresolved`
- `stm_missing_truth_table`
- `stm_truth_table_invalid_row`
- `stm_truth_table_invalid_column`

stm_r_tt_cell_type

Retrieves the data-type of the specified cell in the given truth table.

Function type

```
INTEGER
```

For elements

action	an
activity	ac
subroutine	sb

Syntax

```
stm_r_tt_cell_type (el, row_num, col_num, status)
```

Arguments

Argument	Input/Output	Type	Description
`el`	In	Statemate element	The element ID
`row_num`	In	`INTEGER`	The row number of the cell
`col_num`		`INTEGER`	The column number of the cell
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_id_not_found`
- `stm_unresolved`
- `stm_missing_truth_table`
- `stm_truth_table_invalid_row`
- `stm_truth_table_invalid_column`

Return Values

Although the return value of this function is of type `INTEGER`, the Documentor enables you to reference this value by name. The name is defined internally as a predefined constant in DGL. The possible values are as follows:

- `stm_tt_cell_type_missing`
- `stm_tt_cell_rpn_same_as_down`
- `stm_tt_cell_rpn`
- `stm_tt_cell_dont_care`
- `stm_tt_is_generate_ev`
- `stm_tt_is_not_generate_ev`
- `stm_tt_cell_empty_same_as_up`
- `stm_tt_cell_empty_same_as_up_and_down`
- `stm_tt_is_empty_cell`

stm_r_changes_log

Provides a change log.

Function type

```
LIST OF STRING
```

For elements

chart	ch

Syntax

```
stm_r_changes_log (ch_lst, ascending, per_date, dont_format, status)
```

Arguments

Argument	Input/ Output	Type	Description
ch_lst	In	LIST OF CHART	The list of charts to track.
ascending	In	BOOLEAN	Determines whether the changes are listed in ascending order TRUE).
per_date	In	BOOLEAN	Determines whether the changes are listed chronologically (TRUE).
dont_format	In	BOOLEAN	Determines whether the log file is formatted. If this is TRUE, each log entry is inserted into a returned list element. If it is FALSE, each field of the log entry is inserted into a returned list element.
status	Out	INTEGER	The function status code.

Status Codes

- stm_id_out_of_range
- stm_not_chart_id
- stm_id_not_found
- stm_success

stm_r_xx_chart

Returns the chart ID for the specified element.

Note the following:

- You can call this function without indicating the specific element type, as follows:

  ```
  stm_r_chart (id, status)
  ```

- For compound arrows, this function retrieves the chart only when all the arrow segments are in the same element. Otherwise, it returns the value 0.

Function type

CHART

For elements

a-flow-line (basic)	ba
a-flow-line (compound)	af
action	an
activity	ac
actor	actor
boundary box	bb
condition	co
connector	cn
data-item	di
data-store	ds
event	ev
field	fd
function	fn
information-flow	if
lifeline	ll
local data	ld
m-flow-line (compound)	mf
module	md
router	router
state	st
subroutine	sb
subroutine parameter	sp
transition (basic)	bt
transition (compound)	tr
use case	uc
user-defined type	dt

Syntax

```
stm_r_xx_chart (xx_id, status)
```

Arguments

Argument	Input/Output	Type	Description
xx_id	In	Statemate element	The element ID
status	Out	INTEGER	The function status code

Status Codes

- stm_success
- stm_id_out_of_range
- stm_id_not_found

Example

To write the name of the chart in which state S1 is found, the template would contain the following statements:

```
VARIABLE
    STATE      state_id;
    INTEGER    status1, status2, status3;
           .
           .
state_id := stm_r_st ('S1',status1);
WRITE ('\n Chart name is:', stm_r_ch_name(
   stm_r_st_chart(state_id,status2),status3);
           .
           .
```

stm_r_xx_combinationals

Returns a list of strings. Each element of the list holds one combinational assignment, which is connected to the specified element.

You can call this function without indicating the specific element type, as follows:

```
stm_r_combinationals (id, status)
```

Function type

```
LIST OF STRING
```

For elements

activity	ac
chart	ch

Syntax

```
stm_r_xx_combinationals (xx_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`xx_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_error_in_file`
- `stm_missing_field`
- `stm_missing_label`
- `stm_missing_name`
- `stm_file_not_found`
- `stm_id_not_found`
- `stm_id_out_of_range`
- `stm_illegal_parameter`

stm_r_sb_connected_chart

Returns the ID of the procedural statechart connected to the specified subroutine.

Function type

```
STATEMATE ELEMENT
```

For elements

subroutine	sb

Syntax

```
stm_r_sb_connected_chart (sb_id, status)
```

Arguments

Argument	Input/Output	Type	Description
sb_id	In	Statemate element	The element ID
status	Out	INTEGER	The function status code

Status Codes

- stm_success
- stm_id_not_found
- stm_id_out_of_range
- stm_no_connected_chart

stm_r_xx_containing_fields

Returns the list of union or record elements that contain fields.

You can call this function without indicating the specific element type, as follows:

```
stm_r_containing_fields (id, status)
```

Function type

```
LIST OF FIELDS
```

For elements

data-item	di
user-defined type	dt

Syntax

```
stm_r_xx_containing_fields (xx_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`xx_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_not_found`
- `stm_id_out_of_range`
- `stm_missing_field`
- `stm_unresolved`

stm_r_ch_creation_date

Returns the date (as a string) on which the specified chart was created.

Note: This function is relevant only for charts that were explicitly defined using one of the graphic editors.

Function type

```
STRING
```

For elements

chart	ch

Syntax

```
stm_r_ch_creation_date (ch_id, status)
```

Arguments

Argument	Input/Output	Type	Description
ch_id	In	Statemate element	The element ID
status	Out	INTEGER	The function status code

Status Codes

- stm_success
- stm_id_not_found
- stm_id_out_of_range
- stm_unresolved

Example

To retrieve the chart date, use the following statements:

```
VARIABLE
    CHART      chart_id;
    INTEGER    status;
           .
           .
                .
chart_id := stm_r_ch ('TOP', status);
WRITE ('\n Chart created on:',
   stm_r_ch_creation_date (chart_id, status));
           .
           .
           .
```

The template writes to the document the date on which the chart named TOP was created.

stm_r_ch_creator

Returns the name of the Statemate user who created the specified chart. This function is relevant only for charts that were explicitly created using one of the graphic editors.

Function type

`STRING`

For elements

chart	ch

Syntax

```
stm_r_ch_creator (ch_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`ch_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_id_not_found`
- `stm_unresolved`

Example

To retrieve the name of the user who created the chart, use the following statements:

```
VARIABLE
    CHART      chart_id;
    INTEGER    status;
           .
           .
           .
chart_id := stm_r_ch ('TOP', status);
WRITE ('\n Chart created by:',
   stm_r_ch_creator (chart_id, status));
           .
           .
           .
```

The name written to the document is the name of the user who created the chart named `TOP`.

stm_r_xx_data_type

Returns the element subtype, including its data type and data structure. For example:

```
stm_xx_union_array, stm_xx_integer, stm_xx_real_queue
```

You can call this function without indicating the specific element type, as follows:

```
stm_r_data_type (id, status)
```

Function type

`INTEGER`

For elements

data-item	di
field	fd
local data	ld
subroutine parameter	sp
user-defined type	dt

Syntax

```
stm_r_xx_data_type (xx_id, status)
```

Arguments

Argument	Input/ Output	Type	Description
`xx_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_error_in_file`
- `stm_file_not_found`
- `stm_illegal_parameter`
- `stm_missing_field`

stm_r_rt_date

Returns the date field of the specified requirement record.

Function type

```
STRING stm_date
```

For elements

requirement	rt

Syntax

```
stm_r_rt_date (rt_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`rt_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_error_in_file`
- `stm_file_not_found`
- `stm_id_out_of_range`
- `stm_illegal_parameter`
- `stm_missing_field`

stm_r_xx_definition_type

Returns the definition type of the specified textual element.

Note the following:

- You can call this function without indicating the specific element type, as follows:

  ```
  stm_r_definition_type (id, status)
  ```

- The enumerated type that reflects whether the textual element has a form. The nature of the definition field in the form is `stm_definition_type`, whose values are as follows:
  - `stm_reference`—The element has no form.
  - `stm_primitive`—The definition field is empty.
  - `stm_compound`—The definition field contains a compound expression.
  - `stm_constant`—The definition field contains a constant.
  - `stm_alias`—The definition field contains an identifier, a bit array, a component, or a slice (relevant for `di` only).
  - `stm_explicit`—The `info_flow` has a form.
  - `stm_predefined`—Predefined function.
- Note that these types are not explicitly specified, but derived from the specification.

Function type

```
INTEGER
```

For elements

action	an
condition	co
data-item	di
enumerated value	en
event	ev
function	fn
information-flow	if
local data	ld
subroutine	sb
subroutine parameter	sp

Syntax

```
stm_r_xx_definition_type (xx_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`xx_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_not_found`

Return Values

Although the return value of this function is of type `INTEGER`, the Documentor enables you to reference this value by name. The name is defined internally as a predefined constant in DGL. The following table lists the possible values for each Statemate element sub type:

Element	Abbreviation	Element Sub-Type
action	an	`stm_an_reference`
		`stm_an_primitive`
		`stm_an_compound`
condition	co	`stm_co_reference`
		`stm_co_primitive`
		`stm_co_compound`
		`stm_co_constant`
data-item	di	`stm_di_reference`
		`stm_di_primitive`
		`stm_di_compound`
		`stm_di_constant`
		`stm_di_alias`
event	ev	`stm_ev_reference`
		`stm_ev_primitive`
		`stm_ev_compound`
field	fd	`stm_fd_primitive`
information-flow	if	`stm_if_reference`
		`stm_if_explicit`
local data	ld	`stm_sp_defined`

subroutine	sb	stm_sb_reference
		stm_sb_predefined
		stm_sb_function
		stm_sb_procedure
		stm_sb_task
subroutine parameter	sp	stm_sp_defined
user-defined type	dt	stm_dt_reference
		stm_dt_primitive

stm_r_xx_desc_file

Returns the description file for the specified element.

You can call this function without specifying the element, as follows:

```
stm_r_desc_file (el, status)
```

Function type

`STRING`

For elements

actor	actor
boundary box	bb
use case	uc

Syntax

```
stm_r_xx_desc_file (elem, status)
```

Arguments

Argument	Input/Output	Type	Description
`elem`	In	Statemate element	The element
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_id_out_of_range`
- `stm_missing_description_file`
- `stm_not_use_case`
- `stm_not_actor`
- `stm_not_boundry_box`
- `stm_success`

stm_r_xx_description

Returns the short description of the specified element. The short description is defined in the element's form.

You can call this function without indicating the specific element type, as follows:

```
stm_r_description (id, status)
```

Function type

```
STRING
```

For elements

action	an
activity	ac
actor	actor
boundary box	bb
chart	ch
condition	co
data-item	di
data-store	ds
event	ev
field	fd
information-flow	if
lifeline	ll
local data	ld
module	md
router	router
state	st
subroutine	sb
subroutine parameter	sp
user-defined type	dt

Syntax

```
stm_r_xx_description (xx_id, status)
```

Arguments

Argument	Input/ Output	Type	Description
`xx_id`	In	Statemate element	The element ID.
`status`	Out	`INTEGER`	The function status code. If no description exists in the element's form, status receives the value `stm_missing_short_description`.

Status Codes

- `stm_success`
- `stm_unresolved`
- `stm_id_out_of_range`
- `stm_id_not_found`
- `stm_missing_short_description`

Example

To retrieve the contents of the short description field in the form of state `SSS.S1`, use the following statements:

```
VARIABLE
   STATE              state_id;
   STRING             state_desc;
   INTEGER            status;
                    .
                    .
                    .
state_id  := stm_r_st ('SSS.S1', status);
state_desc:= stm_r_st_description (state_id, status);
                    .
                    .
                    .
```

`state_desc` contains the short description for the state `SSS.S1` (whose ID is `state_id`).

stm_r_design_attr

Retrieves the design attributes of an element in a form of a list of strings. Each element in the list includes a key-value pair.

Function type

```
LIST OF STRING
```

For elements

element ID	

Syntax

```
stm_r_design_attr (el, status)
```

Arguments

Argument	Input/Output	Type	Description
`el`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`

stm_r_xx_displayed_name

Returns the name of a chart, as it appears in the graphic editor where the specified element is located.

You can call this function without indicating the specific element type, as follows:

```
stm_r_displayed_name (id, status)
```

Function type: `STRING`

For elements

activity	ac
data-store	ds
module	md
module-occurrence	om
off-page activity	oa
router	router
state	st

Syntax

```
stm_r_xx_displayed_name (xx_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`xx_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_not_found`
- `stm_id_out_of_range`
- `stm_error_in_file`
- `stm_illegal_parameter`
- `stm_file_not_found`
- `stm_missing_name`
- `stm_missing_field`

Example

To retrieve the name of the chart as it appears in the title bar of the graphic editor (for example, `FUNC PROC EXAMPLE`) and write it to the document, use the following statements:

```
VARIABLE
    STATE         state_id;
    INTEGER       status;
                  .
                  .
                  .
state_id := stm_r_st (’S1.S3’, status);
WRITE (stm_r_st_displayed_name (state_id, status));
                  .
                  .
                  .
```

If this function does not meet your needs, there are two other `name` functions that return different values:

- `stm_r_xx_name`
- `stm_r_xx_uniquename`

stm_r_ddb_list_names

Returns the names of the lists created by the properties browser.

Syntax

```
stm_r_ddb_list_names (&status)
```

Arguments

Argument	Input/Output	Type	Description
status	Out	int INTEGER	The function status code

Status Codes

- stm_success

stm_r_element_type

Returns the element type of the specified element.

Function type

```
INTEGER (predefined constant)
```

For elements

All types

Syntax

```
stm_r_element_type (id, status)
```

Arguments

Argument	Input/Output	Type	Description
`id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_id_not_found`

Return Values

Although the return value of this function is of type `INTEGER`, the Documentor enables you to reference this value by name. The name is defined internally as a predefined constant in DGL. each element type has an associated predefined value, as shown in the following table:

Element Type	Value
`a-flow-line (basic)`	`stm_a_flow_line`
`a-flow-line (compound)`	`stm_compound_a_flow_line`
`action`	`stm_action`
`activity`	`stm_activity`
`chart`	`stm_chart`
`condition`	`stm_condition`
`connector in activity-chart`	`stm_a_connector`
`connector in module-chart`	`stm_m_connector`
`connector in statechart`	`stm_s_connector`

Element Type	Value
data-item	stm_data_item
data-store	stm_data_store
decomposed sequence diagram	stm_decomposed_sd
event	stm_event
external lifeline	stm_external_lifeline
flow label	stm_flow_label
information-flow	stm_information_flow
lifeline	stm_lifeline
m-flow-line (basic)	stm_label
m-flow-line (compound)	stm_m_flow_line
message	stm_message
module	stm_compound_m_flow_line
module occurrence	stm_module
order insignificant	stm_order_insignificant
separator	stm_separator
subroutine	stm_subroutine
state	stm_module_occurrence
timing constraint	stm_timing_constraint
transition (basic)	stm_state
transition (compound)	stm_transition
transition label	stm_compound_transition

Example

The sample template performs the following tasks:

- Generates a list of elements (of type `MIXED`) using `stm_r_mx_in_definition_of_co`. Elements in this list are all elements (not necessarily conditions) appearing in the definition field of the condition `C1`.
- Searches this list for conditions. If any are found, it prints them in the document.

```
VARIABLE
    CONDITION          cond_id;
    LIST OF ELEMENT    elmnt_list;
    ELEMENT            el;
    INTEGER            status, el_type;
                     .
                     .
                     .
cond_id := stm_r_co ('C1', status);
elmnt_list := stm_r_mx_in_definition_of_co
({cond_id}, status);
FOR el IN elmnt_list LOOP
    el_type := stm_r_element_type (el, status);
```

```
        IF el_type = stm_condition THEN
           WRITE ('\n Condition Name:', stm_r_co_name(
              el, status));
        END IF;
    END LOOP;
                        .
                        .
                        .
```

stm_r_xx_expr_hyper

Returns the definition expression of the specified element found in the **Definition** field of the element's form, including hyperlinks to referenced elements.

Function type

`STRING`

For elements

a-flow-lines (basic)	ba
action	an
condition	co
data-item	di
event	ev
message	msg
m-flow-line (basic)	bm
subroutine action language	sb_action_lang
transitions (basic)	bt
user-defined type	dt

Syntax

```
stm_r_xx_expr_hyper (elem, format, status)
```

Arguments

Argument	Input/Output	Type	Description
`elem`	In	Statemate element	The element ID
`format`	In	`STRING`	Either FRAMEMAKER or WORD
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`

- `stm_id_out_of_range`
- `stm_id_not_found`
- `stm_unresolved`
- `stm_primitive_element`

stm_r_xx_expression

Returns the definition expression of the specified element found in the **Definition** field of the element's form. For arrows, this function returns the label attached to the arrow. The function is performed for basic arrows (arrow segments that connect boxes and connectors).

Note the following:

- You can call this function without indicating the specific element type, as follows:

  ```
  stm_r_expression (id, status)
  ```

- This function is valid for compound textual elements, which are defined as an expression using the **Definition** field of its form.

Function type

```
STRING
```

For elements

a-flow-line (basic)	ba
action	an
condition	co
data-item	di
event	ev
message	msg
m-flow-line (basic)	bm
transition (basic)	bt
user-defined type	dt

Syntax

```
stm_r_xx_expression (xx_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`xx_id`	In	Statemate element	The element ID.
`status`	Out	`INTEGER`	The function status code. If `xx_id` belongs to a primitive (not a compound) element, `status` receives the value `stm_primitive_element`.

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_id_not_found`
- `stm_unresolved`
- `stm_primitive_element`

Example

To retrieve the definition of `C1` from the database for a system that contains a condition `C1` (where `C1` is defined as `C2` or `C3` in the form of `C1)`, use the following function calls:

```
VARIABLE
    CONDITION      cond_id;
    STRING         cond_def;
    INTEGER        status;
                   .
                   .
                   .
cond_id := stm_r_co ('C1', status);
cond_def := stm_r_co_expression (cond_id, status);
                   .
                   .
                   .
```

`cond_def` is assigned the string value `"C2 or C3"`.

stm_r_xx_ext_link

Retrieves the external file link for the specified element. You specify the external link by selecting **Edit > Link** in the properties for the element.

You can call this function without specifying an element type, as follows:

```
stm_r_ext_link (elem, status)
```

Function type

```
STRING
```

For elements

action	an
activity	ac
actor	actor
boundary box	bb
chart	ch
condition	co
data-item	di
data-store	ds
event	ev
field	fd
information-flow	if
lifeline	ll
module	md
router	router
state	st
subroutine	sb
use case	uc
user-defined type	dt

Syntax

```
stm_r_xx_ext_link (elem, status)
```

Arguments

Argument	Input/Output	Type	Description
`elem`	In		The Statemate element
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_id_not_found`
- `stm_id_out_of_range`
- `stm_name_not_found`
- `stm_auto_defined`
- `stm_missing_external_link`
- `stm_success`

stm_r_uc_ext_point_def

Retrieves the extension point definitions for the specified use case.

Function type

`STRING`

For elements

use case	uc

Syntax

```
stm_r_uc_ext_point_def (uc, status)
```

Arguments

Argument	Input/Output	Type	Description
uc	In	USE_CASE	The use case
status	Out	INTEGER	The function status code

Status Codes

- `stm_id_out_of_range`
- `stm_not_use_case`
- `stm_missing_extension_point_definition`
- `stm_success`

stm_r_formal_parameter_names

Returns a list of names of formal parameters that appear in bindings of instance boxes and components.

Function type

```
LIST OF STRING
```

For elements

action	an
condition	co
data-item	di
event	ev

Syntax

```
stm_r_formal_parameter_names (inst_box_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`xx_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`

stm_r_sb_global_data

Returns the global data associated with the specified subroutine.

Function type

```
LIST OF ELEMENT
```

For elements

subroutine	sb

Syntax

```
stm_r_sb_global_data (sb_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`xx_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_missing_global_data`
- `stm_id_out_of_range`
- `stm_id_not_found`

stm_r_sb_global_data_mode

Returns the mode of a subroutine's global variable.

Function type

`stm_parameter_mode`

For elements

subroutine	sb

Syntax

```
stm_r_sb_global_data_mode (fn_id, stm_id pd_id, int *status)
```

Arguments

Argument	Input/Output	Type	Description
`fn_id`			The subroutine ID.
`pd_id`			The global variable ID.
`status`	Out	`int`	The function status code.

Status Codes

- `stm_success`
- `stm_missing_global_data`
- `stm_id_out_of_range`
- `stm_id_not_found`

stm_r_global_interface_report

Creates a global interface report from the specified input list.

Function type

```
LIST OF STRING
```

For elements

activity	ac
external router	ext_router
router	router

Syntax

```
stm_r_global_interface_report (elm_lst, sort_by_elm, status)
```

Arguments

Argument	Input/Output	Type	Description
`elm_list`	In	`LIST OF ELEMENT`	The list of elements to include in the report
`sort_by_elm`	In	`BOOLEAN`	Specifies whether to sort the elements in the report by element name (TRUE) or activity name (FALSE)
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_id_out_of_range`
- `stm_success`

stm_r_xx_cbk_binding_expression_hyper

Retrieves the callback binding expressions, with hyperlinks to referenced elements.

You can call this function without indicating the specific element type, as follows:

```
stm_r_cbk_binding_expression_hyper (id, &status)
```

Function Type

```
stm_expression
```

For Elements

activity	ac
condition	co
data-item	di
event	ev
state	st

Syntax

```
stm_r_xx_cbk_binding_expression_hyper (id, char* formator int*status)
```

Arguments

Argument	Input/ Output	Type	Description
xx_id	In	stm_id	The element ID.
format	In	STRING	Either FrameMaker or Word.
status	Out	int	The function status code.

Status Codes

- stm_success
- stm_id_not_found
- stm_id_out_of_range
- stm_missing_cbk_binding
- stm_unresolved

stm_r_xx_graphic

Returns the graphical information associated with the specified element.

Note the following:

- You can call this function without indicating the specific element type, as follows:

  ```
  stm_r_graphic(id, status)
  ```

- Each environment module can have several occurrences with the same name in a chart. Call the query function `stm_r_om_of_md` to get the graphical information of its occurrences, then use the function `stm_r_om_graphic` for each occurrence.

Function type

`STRING`

For elements

activity	ac
basic a-flow-line	ba
basic m-flow-line	bm
basic transition	bt
combinational assignment	ca
connector	cn
data-store	ds
module	md
module-occurence	om
note	nt
off-page activity	oa
state	st

Syntax

```
stm_r_xx_graphic (xx_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`xx_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_unresolved`
- `stm_id_not_found`
- `stm_missing_graphic_data`

Note

When `stm_unresolved` is returned, no record is received.

stm_r_hyper_key

Retrieves the unique key for the specified element.

Function type

`STRING`

For elements

element ID	ac

Syntax

```
stm_r_hyper_key (el, status)
```

Arguments

Argument	Input/Output	Type	Description
`el`	In	`ELEMENT`	The element ID whose key you want
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_id_out_of_range`
- `stm_success`

stm_r_md_implementation

Retrieves the implementation type for the specified module.

Function type

`STRING`

For elements

module	md

Syntax

```
stm_r_md_implementation (md_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`md_id`	In	Statemate element	The module ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_id_not_found`
- `stm_unresolved`
- `stm_not_instance`

stm_r_included_gds

Returns the list of global definition sets contained in the specified chart.

Function type

```
LIST OF CHART
```

For elements

chart	ch

Syntax

```
stm_r_included_gds (ch_id, status)
```

Arguments

Argument	Input/Output	Type	Description
ch_id	In	CHART	The chart
status	Out	INTEGER	The function status code

Status Codes

- stm_id_out_of_range
- stm_use_all_public_gds
- stm_success

stm_r_msg_included_in_ord_insig

Returns a list of messages that are bounded by an order-insignificant element.

Function type

```
LIST OF MESSAGE
```

For elements

Last of order insignificance	ord_insig_list

Syntax

```
stm_r_msg_included_in_ord_insig (ord_insig_list, status)
```

Arguments

Argument	Input/ Output	Type	Description
`ord_insig_list`	In	`LIST OF ORDER_INSIGNIFICANT`	A list of elements
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_not_order_insignificant`

stm_r_cd_info

Retrieves the description of the specified continuous chart.

Function type

```
STRING
```

For elements

chart	ch

Syntax

```
stm_r_cd_info (ch, status)
```

Arguments

Argument	Input/Output	Type	Description
cd	In	ChART	The chart
status	Out	INTEGER	The function status code

Status Codes

- stm_id_out_of_range
- stm_null_string
- stm_success

stm_r_inherited_gds

Retrieves the list of global definition sets that are “inherited” (included indirectly) by the specified chart.

Function type

```
LIST OF CHART
```

For elements

chart	ch

Syntax

```
stm_r_inherited_gds (ch_id, status)
```

Arguments

Argument	Input/Output	Type	Description
ch_id	In	CHART	The chart
status	Out	INTEGER	The function status code

Status Codes

- stm_id_out_of_range
- stm_use_all_public_gds
- stm_success

stm_r_xx_instance_name

Returns the name of the instance as it appears in the chart for a specific hierarchical Statemate element.

Note the following:

- You can call this function without indicating the specific element type, as follows:

  ```
  stm_r_instance_name (id, status)
  ```

- This function is relevant only for states, internal modules, and regular or control activities, because only these elements can have instances.

Function type

```
STRING
```

For elements

activity	ac
module	md
state	st

Syntax

```
stm_r_xx_instance_name (xx_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`xx_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_id_not_found`
- `stm_unresolved`
- `stm_not_instance`

Example

To retrieve the name of an instance for state named `S1@S1_def`, use the following statements:

```
VARIABLE
    STATE           state_id;
    INTEGER         status;
                    .
                    .
                    .
state_id := stm_r_st ('S1', status);
WRITE ('\n Instance  Name:',
  stm_r_st_instance_name (state_id, status));
                    .
                    .
                    .
```

The name written to the document is `S1@S1_DEF`.

stm_r_xx_keyword

Retrieves a portion of the element's long description. An element's long description is attached to its form.

You can call this function without indicating the specific element type, as follows:

```
stm_r_keyword (id, begin_keyword, end_keyword, filename, status)
```

Function type

`STRING` (a file name)

For elements

action	an
activity	ac
actor	actor
boundary box	bb
chart	ch
condition	co
data-item	di
data-store	ds
event	ev
field	fd
information-flow	if
lifeline	ll
module	md
router	router
state	st
subroutine	sb
use case	uc
user-defined type	dt

Syntax

```
stm_r_xx_keyword (xx_id, begin_keyword, end_keyword, filename, status)
```

Arguments

Argument	Input/Output	Type	Description
`xx_id`	In	`stm_id`	The element ID
`begin_< keyword`	In	`char *`	The beginning of the portion of the string in the long description to extract
`end_keyword`	In	`char *`	The end of the portion of the string in the long description to extract
`filename`	In	`stm_filename`	The name of the file to contain the long description
`status`	Out	`int`	The function status code

Note the following:

- The arguments `begin_keyword` and `end_keyword` are strings of text appearing in the element's long description. The portion extracted from the database begins with the line following `begin_keyword` and extends to the line preceding `end_keyword`.
- If the value of `begin_keyword` does not appear in the long description, the function creates an empty file; `status` then receives the value `stm_starting_keyword_not_found`.
- If the value of `end_keyword` does not appear in the long description, the entire long description (from the line following the value of `begin_keyword`) is retrieved; `status` receives the value `stm_ending_keyword_not_found`.
- The values of `begin_keyword` and `end_keyword` must appear at the beginning of a line in the long description.
- `filename` follows the conventions of the operating system. It returns the value of the argument `filename` (when one is specified). If an empty string `''` (two contiguous quotation marks) is specified for `filename`, Statemate creates a temporary file where it stores the text. The name of this temporary file is returned by this function.
- If no long description exists for the element, `status` receives the value `stm_missing_long_description`.

Status Codes

- stm_success
- stm_unresolved
- stm_id_out_of_range
- stm_id_not_found
- stm_can_not_open_file
- stm_name_not_found
- stm_missing_long_description
- stm_starting_keyword_not_found
- stm_ending_keyword_not_found

Example

The long description for the state WAIT contains the following section:

```
!BHV_DESCR
When the assembly process reaches the critical stage where all parts
must be carefully selected, mounted and assembled, we wait for the
interrupt signal to tell us that all the required parts are in place
before continuing. This state acts as a synchronization point in the
assembly process.
!END_DESCR
```

To extract the portion of the long description beginning with “When the...” and ending with “... assembly process,” use the following function call:

```
VARIABLE
    STATE           state_id;
    STRING          descr_file;
    INTEGER         status;
                     .
                     .
                     .
state_id:=stm_r_st ('WAIT', status);
descr_file:=stm_r_st_keyword (state_id,'!BHV_DESCR',
                        '!END_DESCR','',status);
INCLUDE (descr_file);
                     .
                     .
                     .
```

The portion of the long description is written to a temporary file and written to your document.

stm_r_sb_kr_c_user_code

Returns the K&R C code that was manually written by the user for the specified subroutine.

Function type

```
LIST OF STRING
```

For elements

subroutine	sb

Syntax

```
stm_r_sb_kr_c_user_code (sb_id, status)
```

Arguments

Argument	Input/Output	Type	Description
sb_id	In	Statemate element	The element ID
status	Out	INTEGER	The function status code

Status Codes

- stm_success
- stm_id_out_of_range
- stm_missing_user_code

stm_r_xx_labels

Returns a list of strings that consists of all the labels of the specified compound transition or message. The labels appear on the transition segments that comprise the specified compound transition, or on the message. The syntax of these labels is `trigger/action`.

Note: To divide the labels into their trigger and action parts, use the utility routines `stm_trigger_of_reaction` and `stm_action_of_reaction`.

Function type

```
LIST OF STRING
```

For elements

message	msg
transition	tr

Syntax

```
stm_r_xx_labels (tr_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`xx_id`	In	`TRANSITION ELEMENT` or `MESSAGE`	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_unresolved`
- `stm_id_out_of_range`
- `stm_id_not_found`
- `stm_missing_label`

Example

To extract all the labels of transitions exiting from state `S1`, the template should contain the following statements:

```
VARIABLE
    STATE               state_id
    INTEGER             status;
    LIST OF STRING      labels;
    LIST OF TRANSITION  trans;
    TRANSITION          tr;
    STRING              lab;
                       .
                       .
  state_id := stm_r_st ('S1',status);
  trans := stm_r_tr_from_source_st ({state_id}, status);
  FOR tr IN trans LOOP
    labels:=stm_r_tr_labels(tr, status);
    IF status = stm_success THEN
      FOR lab IN labels LOOP
                       .
                       .
                       .
      END FOR;
    END IF;
  END FOR;
```

stm_r_xx_labels_hyper

Returns a list of strings of message or transition labels, with hyperlinks to referenced elements.

Function type

```
LIST OF STRING
```

For elements

message	msg
transition	tr

Syntax

```
stm_r_xx_labels_hyper (message, format, status)Arguments
```

Argument	Input/Output	Type	Description
elem_id	In	Statemate element	The element ID
format	In	STRING	Either FrameMaker or Word
status	Out	INTEGER	The function status code

Status Codes

- stm_success
- stm_id_out_of_range
- stm_id_not_found
- stm_unresolved

stm_r_local_interface_report

Creates a local interface report from the specified input list.

Function type

```
LIST OF STRING
```

For elements

activity	ac
external router	ext_router
router	router

Syntax

```
stm_r_local_interface_report (elm_lst, status)
```

Arguments

Argument	Input/Output	Type	Description
`elm_list`	In	`LIST OF ELEMENT`	The list of elements to include in the report
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_id_out_of_range`
- `stm_success`

stm_r_xx_longdes

Retrieves the long description attached to the specified element.

You can call this function without indicating the specific element type, as follows:

```
stm_r_longdes (id, filename, status)
```

Function type

```
STRING (file name)
```

For elements

action	an
activity	ac
actor	actor
boundary box	bb
chart	ch
condition	co
data-item	di
data-store	ds
event	ev
field	fd
information-flow	if
lifeline	ll
module	md
router	router
state	st
subroutine	sb
use case	uc
user-defined type	dt

Syntax

```
stm_r_xx_longdes (xx_id, filename, status)
```

Arguments

Argument	Input/Output	Type	Description
xx_id	In	Statemate element	The element ID
filename	In	stm_filename STRING	The name of the file that will contain the long description
status	Out	INTEGER	The function status code

Note the following:

- The filename follows the conventions of the host operating system.
- This function returns the value of the argument filename when one is specified. If an empty string ''(two contiguous quotation marks) is specified, Statemate creates a temporary file where it stores the text. The name of this temporary file is returned by the function.
- If no long description exists for the element, status receives the value stm_missing_long_description.

Status Codes

- stm_unresolved
- stm_success
- stm_id_out_of_range
- stm_id_not_found
- stm_can_not_open_file
- stm_missing_long_description

Example

To retrieve the long description for the activity A1, use the following statements:

```
VARIABLE
    ACTIVITY          act_id;
    STRING            long_des_file;
    INTEGER           status;
                      .
                      .
act_id := stm_r_ac ('A1',status);
long_des_file:=
  stm_r_ac_longdes(act_id,'text.txt',status);
                      .
                      .
```

The long description for activity A1 is written to the file text.txt. This file resides in the current working directory. The variable long_des_file contains the string 'text.txt'.

stm_r_lookup_table_header

Retrieves the header for the lookup table.

Function type

```
LIST OF STRING
```

For elements

lookup table	lookup table

Syntax

```
stm_r_lookup_table_header (el, status)
```

Arguments

Argument	Input/Output	Type	Description
`el`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`

stm_r_xx_max_val

Returns the maximum value of the specified element.

You can call this function without indicating the specific element type:

```
stm_r_max_val (id, status)
```

Function type

```
STRING
```

For elements

data-item	di
field	fd
local data	ld
subroutine parameter	sp
user-defined type	dt

Syntax

```
stm_r_xx_max_val (xx_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`xx_id`	In	`Statemate element`	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_not_found`
- `stm_id_out_of_range`

stm_r_xx_min_val

Returns the minimum value of the specified element.

You can call this function without indicating the specific element type, as follows:

```
stm_r_min_val (id, status)
```

Function type

```
STRING
```

For elements

data-item	di
field	fd
local data	ld
subroutine parameter	sp
user-defined type	dt

Syntax

```
stm_r_xx_min_val (xx_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`xx_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_not_found`
- `stm_id_out_of_range`

stm_r_xx_mini_spec

Returns a string with mini-spec reactions or actions.

You can call this function without indicating the specific element type, as follows:

```
stm_r_mini_spec (id, status)
```

Function type

`STRING`

For elements

activity	ac

Syntax

```
stm_r_xx_mini_spec (xx_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`xx_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_id_not_found`
- `stm_unresolved`
- `stm_missing_label`

stm_r_ac_mini_spec_hyper

Returns a string with the mini-spec, including hyperlinks to referenced elements.

Function type

`STRING`

For elements

activity	ac

Syntax

```
stm_r_ac_mini_spec_hyper (elem, format, status)
```

Arguments

Argument	Input/Output	Type	Description
`elem`	In	Statemate element	The element ID
`format`	In	`STRING`	Either FrameMaker or Word
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_id_not_found`
- `stm_unresolved`

stm_r_xx_mode

Returns the parameter or router mode.

Function type

`INTEGER`

For elements

parameter	parameter

Syntax

```
stm_r_xx_mode (elem_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`elem_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_id_not_found`
- `stm_unresolved`

Retrun Values

Although the return value of this function is of type `INTEGER`, the Documentor enables you to reference this value by name. The name is defined internally as a predefined constant in DGL.

stm_r_ch_modification_date

Returns the date in which the version of the chart in the workarea was saved in the databank. This function is relevant only for charts that were explicitly defined using one of the graphics editors.

Function type

`STRING`

For elements

chart	ch

Syntax

```
stm_r_ch_modification_date (ch_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`ch_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_id_not_found`
- `stm_unresolved`

Example

To retrieve the date of the last modification for a chart, use the following statements:

```
VARIABLE
    CHART           chart_id;
    INTEGER         status;
                    .
                    .
                    .
chart_id := stm_r_ch ('TOP', status);
WRITE ('\n Chart modified on:',
   stm_r_ch_modification_date (chart_id, status));
                    .
                    .
                    .
```

The template writes to the document the date on which the chart named `TOP` was last modified.

stm_r_xx_name

Returns the element name. For hierarchical elements, the function returns the name associated with the box. Because hierarchical elements can share the same name, the return value does not necessarily uniquely identify an element. To return a unique name, use the function `stm_r_xx_uniquename`.

Note the following:

- This function returns a pointer to a static area of memory. Subsequent calls to this procedure will overwrite the old string. If the name needs to be preserved, use the `strdup()` function from the string library.
- You can call this function without indicating the specific element type, as follows:

  ```
  stm_r_name (id, status)
  ```

- For boxes that have no names, this function returns the definition chart name. For example, for box `@ABC`, this function returns `ABC`.

Function type

```
STRING
```

For elements

action	an
activity	ac
actor	actor
boundary box	bb
chart	ch
condition	co
data-item	di
data-store	ds
enumerated value	en
event	ev
field	fd
function	fn
information-flow	if
lifeline	ll
local data	ld
module	md
router	router
state	st
subroutine	sb
subroutine parameter	sp
use case	uc
user-defined type	dt

Syntax

```
stm_r_xx_name (xx_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`xx_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- stm_success
- stm_error_in_file
- stm_id_out_of_range
- stm_id_not_found
- stm_missing_name
- stm_missing_field
- stm_illegal_parameter
- stm_file_not_found

Example

To retrieve and print the name of a state in a statechart, use the following statements:

```
VARIABLE
    STATE       state_id;
    INTEGER     status;
                .
                .
state_id := stm_r_st (’S1.S3’,status);
WRITE (stm_r_st_name (state_id, status));
                .
                .
```

In this example, the state name is provided and this value is used to retrieve the same state name from the database. The purpose of this example is to demonstrate the value returned by this function, in contrast to the value returned by the function `stm_r_xx_uniquename`.

stm_r_next_msg

Returns the message after (in time) the decomposed sequence diagram.

Function type

```
MESSAGE
```

For elements

decomposed SD	dec_sd

Syntax

```
stm_r_next_msg (dec_sd_id, status)
```

Arguments

Argument	Input/Output	Type	Description
dec_sd_id	In	REFERENCED_SD	The element ID
status	Out	INTEGER	The function status code

Status Codes

- stm_success
- stm_id_out_of_range
- stm_not_decomposed_sd
- stm_message_not_found

stm_r_xx_note

Returns the note for the specified element in the sequence diagram.

Function type

```
LIST OF STRING
```

For elements

separator	sep
order insignificant line	ord_insig
message	msg
timing constraint	tc

Syntax

```
stm_r_xx_note (tr_id, status)
```

Arguments

Argument	Input/ Output	Type	Description
`tr_id`	In	Statemate element	The transition ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_id_out_of_range`
- `stm_missing_note`
- `stm_success`

stm_r_xx_notes

Returns the note for the specified input transition.

Function type

```
LIST OF STRING
```

For elements

chart	ch
transistion	tr

Syntax

```
stm_r_xx_notes (tr_id, status)
```

Arguments

Argument	Input/ Output	Type	Description
`tr_id`	In	Statemate element	The transition ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_id_out_of_range`
- `stm_missing_note`
- `stm_success`

stm_r_tt_num_of_col

Retrieves the number of columns (including blank ones) in the specified truth table, as viewed in the truth table editor.

Function type

```
INTEGER
```

For elements

truth table	tt

Syntax

```
function stm_r_tt_num_of_col (el, status)
```

Arguments

Argument	Input/Output	Type	Description
el	In	ELEMENT	The element ID
status	Out	INTEGER	The function status code

Status Codes

- stm_missing_truth_table
- stm_success

stm_r_tt_num_of_in

Retrieves the number of input columns in the specified truth table.

Function type

```
INTEGER
```

For elements

truth table	tt

Syntax

```
stm_r_tt_num_of_in (el_id, status)
```

Arguments

Argument	Input/Output	Type	Description
el_id	In	ELEMENT	The element ID
status	Out	INTEGER	The function status code

Status Codes

- stm_missing_truth_table
- stm_success

stm_r_tt_num_of_out

Retrieves the number of output columns in the specified truth table.

Function type

`INTEGER`

For elements

truth table	tt

Syntax

```
stm_r_tt_num_of_out (el_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`el_id`	In	`ELEMENT`	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_missing_truth_table`
- `stm_success`

stm_r_tt_num_of_row

Retrieves the number of rows (including blank ones) in the specified truth table, as viewed in the truth table editor.

Function type

```
INTEGER
```

For elements

truth table	tt

Syntax

```
stm_r_tt_num_of_row (el_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`el_id`	In	`ELEMENT`	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_missing_truth_table`
- `stm_success`

stm_r_uc_num_of_scen

Retrieves the number of scenarios for the specified use case.

Function type

`INTEGER`

For elements

use case	uc

Syntax

```
function stm_r_uc_num_of_scen (uc, status)
```

Arguments

Argument	Input/Output	Type	Description
uc	In	`USE_CASE`	The use case
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`

stm_r_xx_number_of_bits

Returns the number of bits in the element.

You can call this function without indicating the specific element type, as follows:

```
stm_r_number_of_bits (id, status)
```

Function type

```
STRING
```

For elements

data-item	di
field	fd
local data	ld
subroutine parameter	sp
user-defined type	dt

Syntax

```
stm_r_xx_number_of_bits (xx_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`xx_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_not_found`
- `stm_id_out_of_range`

stm_r_xx_of_enum_type

Retrieves the enumerated type ID (a user-defined type) for the specified element.

You can call this function without indicating the specific element type, as follows:

```
stm_r_of_enum_type (id, status)
```

Function type

`DATA_TYPE`

For elements

data-item	di
field	fd
local data	ld
subroutine parameter	sp
user-defined type	dt

Syntax

```
stm_r_xx_of_enum_type (xx_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`xx_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_id_not_found`
- `stm_missing_of_enum_type`

stm_r_xx_of_enum_type_name_type

Retrieves the enumerated name type for the specified elements.

You can call this function without indicating the specific element type, as follows:

```
stm_r_of_enum_type_name_type (id, status)
```

Function type

```
INTEGER (predefined constant)
```

For elements

data-item	di
field	fd
local data	ld
subroutine parameter	sp
user-defined type	dt

Syntax

```
stm_r_xx_of_enum_type_name_type (xx_id, status)
```

Arguments

Argument	Input/Output	Type	Description
xx_id	In	Statemate element	The element ID
status	Out	INTEGER	The function status code

Status Codes

- stm_success
- stm_id_out_of_range
- stm_id_not_found
- stm_missing_of_enum_type

Return Types

Although the return value of this function is of type `INTEGER`, the Documentor enables you to reference this value by name. The name is defined internally as a predefined constant in DGL. The possible values are as follows:

- `stm_ntc_name`
- `stm_ntc_synonym`
- `stm_ntc_unknown`

stm_r_ord_insig_defined_in_ch

Returns the list of order-insignificant elements for the specified charts.

Function type

```
LIST OF ORDER_INSIGNIFICANT
```

For elements

chart	ch

Syntax

```
stm_r_ord_insig_defined_in_ch (ch_lst, status)
```

Arguments

Argument	Input/Output	Type	Description
`ch_lst`	In	`LIST OF CHART`	The list of charts
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_not_order_insignificant`

stm_r_parameter_binding

Returns the parameter expression from generic charts and components.

Function type

```
STRING
```

Syntax

```
stm_r_parameter_binding (xx_paramid_in_gen, inst_boxid, status)
```

Arguments

Argument	Input/Output	Type	Description
`xx_paramid_in_gen`	In	Statemate element	The element ID
`inst_boxid`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_param_not_compatible`
- `stm_name_not_found`
- `stm_not_a_parameter`

stm_r_parameter_mode

Retrieves the parameter mode, including subroutine parameters and the parameters of generic charts and components.

You can call this function without indicating the specific element type, as follows:

```
stm_r_parameter_mode (xx_id, status)
```

Function type

`STRING`

For elements

chart	ch
subroutine parameter	sp

Syntax

```
stm_r_xx_parameter_mode (xx_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`xx_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_not_a_parameter`

Return Values

Although the return value of this function is of type `INTEGER`, the Documnetor enables you to reference this value by name. The name is defined internally as a predefined constant in DGL. The possible values are as follows:

- `stm_in_parameter`
- `stm_out_parameter`
- `stm_inout_parameter`
- `stm_constant_parameter`

stm_r_sb_parameters

Retrieves the parameters of the subroutine.

Function type

```
LIST OF SUBROUTINE PARAMETERS
```

For elements

subroutine	sb

Syntax

```
stm_r_sb_paramaters (sb_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`sb_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_not_a_parameter`
- `stm_missing_subroutine_params`

stm_r_en_parent

Returns the parent type of the specified enumerated value.

Function type

`DATA_TYPE`

For elements

enumerated value	en

Syntax

```
stm_r_en_parent (en, status)
```

Arguments

Argument	Input/Output	Type	Description
`en`	In	• `ELEMENT`	The enumerated value
`status`	Out	• `INTEGER`	The function status code

Status Codes

- `stm_success`

stm_r_previous_msg

Returns the message previous (in time) to the decomposed sequence diagram.

Function type

`MESSAGE`

For elements

decomposed SD	dec_sd

Syntax

```
stm_r_previous_msg (dec_sd_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`dec_sd_id`	In	`REFERENCED_SD`	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_not_decomposed_sd`
- `stm_message_not_found`

stm_r_sb_proc_sch_local_data

Retrieves the local data of the procedural statechart implemented by the specified subroutine.

Function type

```
LIST OF LOCAL_DATA
```

For elements

subroutine	sb

Syntax

```
stm_r_sb_proc_sch_local_data (sb_id, status)
```

Arguments

Argument	Input/Output	Type	Description
sb_id	In	Statemate element	The element ID
status	Out	INTEGER	The function status code

Status Codes

- stm_success
- stm_missing_local_data
- stm_no_connected_chart

stm_r_md_purpose

Returns the purpose of the module.

Function type

`INTEGER`

For elements

module	md

Syntax

```
stm_r_md_purpose (id, status)
```

Arguments

Argument	Input/Output	Type	Description
`id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_not_found`
- `stm_id_out_of_range`
- `stm_unresolved`
- `stm_not_instance`

stm_r_xx_reactions

Returns the static reactions of the specified state. The syntax of these reactions is `trigger/action`.

Note the following:

- To divide the static reactions into their trigger and action parts, use the utility routines `stm_trigger_of_reaction and stm_action_of_reaction`.
- You can call this function without indicating the specific element type, as follows:

```
stm_r_reactions (st_id, status)
```

Function type

```
LIST OF STRING
```

For elements

activity	ac
state	st

Syntax

```
stm_r_xx_reactions (xx_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`xx_st_id`	In	Statemate element	The state ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_unresolved`
- `stm_id_out_of_range`
- `stm_id_not_found`
- `stm_missing_label`

Example

To extract all static reactions of state `S1`, use the following statements:

```
VARIABLE
    STATE            state_id
    INTEGER          status;
    LIST OF STRING   reactions;
    STRING           react;
                       .
                       .
  state_id := stm_r_st ('S1', status);
  reactions := stm_r_st_reactions (state_id, status);
  IF status = stm_success THEN
     FOR react IN reactions LOOP;
                       .
                       .
                       .
```

stm_r_param_binding_hyper

Retrieves the generic instance actual parameter.

Function type

```
STRING
```

Syntax

```
stm_r_param_binding_hyper (prm_id, ins, status)
```

Arguments

Argument	Input/Output	Type	Description
`prm_id`	In	Statemate element	The element ID
`ins`	In	Statemate element	The element ID.
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_id_not_found`
- `stm_unresolved`

stm_r_param_binding_id

Retrieves the Generic Instance actual parameter ID.

Function type

```
STRING
```

Syntax

```
stm_r_param_binding_id (prm_id, ins, status)
```

Arguments

Argument	Input/Output	Type	Description
prm_id	In	Statemate element	The element ID
ins	In	Statemate element	The element ID.
status	Out	INTEGER	The function status code

Status Codes

- stm_success
- stm_id_out_of_range
- stm_id_not_found
- stm_unresolved

stm_r_sb_return_type

Retrieves the subroutine's return type.

Function type

```
INTEGER (predefined constant)
```

For elements

subroutine	sb

Syntax

```
stm_r_sb_return_type (sb_id, status)
```

Arguments

Argument	Input/Output	Type	Description
sb_id	In	Statemate element	The element ID
status	Out	INTEGER	The function status code

Status Codes

- stm_success
- stm_id_out_of_range
- stm_id_not_found

Return Values

Although the return value of this function is of type INTEGER, the Documentor enables you to reference this value by name. The name is defined internally as a predefined constant in DGL.

stm_r_sb_return_user_type

Provide user status.

Function type

`DATA_TYPE`

For elements

subroutine	sb

Syntax

```
stm_r_sb_return_user_type (sb_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`sb_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_id_not_found`
- `stm_missing_user_type`

stm_r_sb_return_user_type_name

Retrieves the subroutine's return user type and name type.

Function type

```
INTEGER (predefined constant)
```

For elements

subroutine	sb

Syntax

```
stm_r_sb_return_user_type_name_type (sb_id, status)
```

Arguments

Argument	Input/Output	Type	Description
sb_id	In	Statemate element	The element ID
status	Out	INTEGER	The function status code

Status Codes

- stm_success
- stm_id_out_of_range
- stm_id_not_found
- stm_missing_user_type

Return Values

Although the return value of this function is of type INTEGER, the Documentor enables you to reference this value by name. The name is defined internally as a predefined constant in DGL. The possible values are as follows:

- stm_ntc_name
- stm_ntc_synonym
- stm_ntc_unknown

stm_r_tt_row

Returns a list of strings that represents a row in the truth table. Each string in the list includes the text in the truth table cell. The row's index range is `[0..num_of_rows-1]`. Row 0 returns the list of table header strings.

Function type

```
LIST OF STRING
```

For elements

truth table	tt

Syntax

```
stm_r_tt_row (el, row_num, status)
```

Arguments

Argument	Input/Output	Type	Description
`el`	In	Statemate element	The element ID
`row_num`	In	`INTEGER`	The row number to retrieve
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_truth_table_invalid_row`
- `stm_missing_truth_table`
- `stm_success`

stm_r_uc_scen

Retrieves the scenario, based on the specified index number.

Function type

```
LIST OF STRING
```

For elements

use case	uc

Syntax

```
stm_r_uc_scen (uc, scen_index, status)
```

Arguments

Argument	Input/Output	Type	Description
uc	In	USER_CASE	The use case diagram
scen_index	In	INTEGER	The index for the scenario
status	Out	INTEGER	The function status code

Status Codes

- stm_success

stm_r_uc_scen_attr_name

Returns the names of attributes associated with the specified use case scenario. Attributes are associated with elements via element forms.

Function type

```
LIST OF STRING
```

For elements

use case	uc

Syntax

```
stm_r_uc_scen_attr_name (uc, scen_index, status)
```

Arguments

Argument	Input/Output	Type	Description
uc	In	USE_CASE	The use case.
scen_index	In	INTEGER	The index for the scenario.
status	Out	INTEGER	The function status code. If no attributes exist for the specified element, status receives the value stm_attribute_name_not_found.

Status Codes

- stm_success
- stm_attribute_name_not_found
- stm_id_not_found
- stm_id_out_of_range
- stm_unresolved

stm_r_uc_scen_attr_val

Retrieves attribute values associated with a particular attribute name for the specified use case scenario.

Function type

```
LIST OF STRING
```

For elements

use case	uc

Syntax

```
stm_r_uc_scen_attr_val (uc, scen_index, attr_name, status)
```

Arguments

Argument	Input/ Output	Type	Description
uc	In	`USE_CASE`	The use case.
`scen_index`	In	`INTEGER`	The index of the scenario.
`attr_name`	In	`STRING`	The attribute name. The attribute name is not case-sensitive.
`status`	Out	`INTEGER`	The function status code. If `attr_name` does not exist for the specified element, `status` receives the value `stm_attribute_name_not_found`.

Note the following:

- Attribute values might exist for attributes with no name. Therefore, if you supply contiguous apostrophes (`''`) for `attr_name`, you retrieve all values for unnamed attributes.
- In most cases, attributes have only one value. However, there are some cases where more than one attribute value is simultaneously meaningful. For example, a module has an attribute `implementation`. The attributes `software` and `hardware` might both be meaningful for some modules. Therefore, Statemate provides the capability of assigning multiple values to attributes, and the function returns a list of these values. When there is a single value, the list consists of one component.

Status Codes

- `stm_success`
- `stm_attribute_name_not_found`
- `stm_id_not_found`
- `stm_id_out_of_range`
- `stm_illegal_name`
- `stm_unresolved`

stm_r_sd_scope

Retrieves the scope of the specified sequence diagram.

Function type

```
CHART
```

For elements

sequence diagram	sd

Syntax

```
stm_r_sd_scope (sd, status)
```

Arguments

Argument	Input/Output	Type	Description
`sd`	In	`CHART`	The sequence diagram
`status`	Out	`INTEGER`	The function status code

stm_r_xx_select_implementation

Retrieves the implementation type of the specified element.

Function type

```
INTEGER
```

For elements

action	an
activity	ac
subroutine	sb

Syntax

```
stm_r_xx_select_implementation (xx_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`xx_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_out_of_range`
- `stm_id_not_found`

Return Values

Although the return value of this function is of type `INTEGER`, the Documentor enables you to reference this value by name. The name is defined internally as a predefined constant in DGL. The possible values are as follows:

- `stm_sb_action_lang`
- `stm_sb_procedural_sch`
- `stm_sb_kr_c_code`
- `stm_sb_ansi_c_code`
- `stm_sb_ada_code`
- `stm_sb_vhdl_code`
- `stm_sb_verilog_code`

- stm_sb_truth_table_code
- stm_sb_best_match
- stm_sb_none

stm_r_st_static_reactions

Returns the static reactions defined for the specified state element.

Function type

STRING

For elements

state	st

Syntax

```
stm_r_st_static_reactions (st_id, status)
```

Arguments

Argument	Input/Output	Type	Description
st_id	In	Statemate element	The element ID
status	Out	INTEGER	The function status code

Status Codes

- stm_success
- stm_id_not_found
- stm_id_out_of_range
- stm_unresolved
- stm_missing_label

stm_r_st_static_reactions_hyper

Returns a string with the static reactions, including hyperlinks to referenced elements.

Function type

`STRING`

For elements

state	st

Syntax

```
stm_r_st_static_reactions_hyper (elem, format, status)
```

Arguments

Argument	Input/Output	Type	Description
`elem`	In	Statemate element	The element ID
`format`	In	• `STRING`	Either FrameMaker or Word
`status`	Out	• `INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_id_not_found`
- `stm_unresolved`

stm_r_xx_string_length

Retrieves the string length of the specified element.

You can call this function without indicating the specific element type, as follows:

```
stm_r_string_length (id, status)
```

Function type

`STRING`

For elements

data-item	di
field	fd
local data	ld
subroutine parameter	sp
user-defined type	dt

Syntax

```
stm_r_xx_string_length (xx_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`xx_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_missing_field`
- `stm_illegal_parameter`
- `stm_file_not_found`
- `stm_error_in_file`

stm_r_xx_structure_type

Returns the structure or type of the specified textual element. The structure or type can be single, array, or queue.

You can call this function without specifying an element type, as follows:

```
stm_r_structure_type (id, status)
```

Function type

`INTEGER`

For elements

condition	co
data-item	di
event	ev
field	fd
local data	ld
subroutine parameter	sp
user-defined type	dt

Syntax

```
stm_r_xx_structure_type (xx_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`xx_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_id_not_found`

Return Values

Although the return value of this function is of type `INTEGER`, the Documentor enables you to reference this value by name. The name is defined internally as a predefined constant in DGL. The following are all possible values allowed for each Statemate element subtype:

Element Type	Element Subtype
condition	`stm_co_array`
	`stm_co_missing`
	`stm_co_single`
data-item	`stm_di_array`
	`stm_di_queue`
	`stm_di_single`
event	`stm_ev_array`
	`stm_ev_missing`
	`stm_ev_single`
field	`stm_fd_array`
	`stm_fd_queue`
	`stm_fd_single`
local data	`stm_ld_array`
	`stm_ld_queue`
	`stm_ld_single`
subroutine parameter	`stm_sp_array`
	`stm_sp_queue`
	`stm_sp_single`
user-defined type	`stm_dt_array`
	`stm_dt_queue`
	`stm_dt_single`

stm_r_ac_subroutine_bind

Returns the subroutine binding connected to the specified activity.

Function type

```
LIST OF STRING
```

For elements

activity	ac

Syntax

```
stm_r_ac_subroutine_bind (ac_id, status)
```

Arguments

Argument	Input/Output	Type	Description
ac_id	In	Statemate element	The element ID
status	Out	INTEGER	The function status code

Status Codes

- stm_success
- stm_id_out_of_range
- stm_unresolved
- stm_id_not_found
- stm_missing_subroutine_binding

stm_r_ac_subroutine_bind_enable

Determines whether the subroutine bound to the specified activity is enabled or disabled.

Function type

`INTEGER`

For elements

activity	ac

Syntax

```
stm_r_ac_subroutine_bind_enable (ac_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`ac_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_unresolved`
- `stm_id_not_found`

Return Values

Although the return value of this function is of type `INTEGER`, the Documentor enables you to reference this value by name. The name is defined internally as a predefined constant in DGL. The possible values are as follows:

- `stm_ac_cbk_enable`
- `stm_ac_cbk_disable`
- `stm_ac_cbk_bind_missing`

stm_r_ac_subroutine_bind_expr

Returns the subroutine binding expression that is connected to the specified activity.

Function type

`STRING`

For elements

activity	ac

Syntax

```
stm_r_ac_subroutine_bind_expr (ac_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`ac_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_id_not_found`
- `stm_unresolved`
- `stm_missing_subroutine_binding`

stm_r_xx_synonym

Retrieves the synonym of the specified element. The synonym is defined in the element's form.

You can call this function without indicating the specific element type, as follows:

```
stm_r_synonym (id, status)
```

Function type

```
STRING
```

For elements

action	an
activity	ac
actor	actor
boundary box	bb
chart	ch
condition	co
data-item	di
data-store	ds
event	ev
field	fd
information-flow	if
lifeline	ll
module	md
router	router
state	st
subroutine	sb
use case	uc
user-defined type	dt

Syntax

```
stm_r_xx_synonym (xx_id, status)
```

Arguments

Argument	Input/Output	Type	Description
xx_id	In	Statemate element	The element ID.
status	Out	INTEGER	The function status code. If no synonym is defined in the element's form, status receives the value stm_missing_synonym.

Status Codes

- stm_success
- stm_unresolved
- stm_missing_subroutine_params
- stm_id_out_of_range
- stm_id_not_found
- stm_missing_synonym

Example

To write out the synonym of activity A1, use the following statements:

```
VARIABLE
     ACTIVITY            act_id;
     INTEGER             status;
                         .
                         .
                         .
act_id := stm_r_ac('A1', status);
WRITE ('\n Synonym:', stm_r_ac_synonym
      (act_id, status));
                         .
                         .
                         .
```

stm_r_ac_termination

Returns the activity termination type specified in the activity form.

Function type

`INTEGER` (predefined constant)

For elements

activity	ac

Syntax

```
stm_r_ac_termination (act_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`act_id`	In	`ACTIVITY` or **ELEMENT**	The activity whose termination type you want to retrieve
`status`	Out	`INTEGER`	Function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_id_not_found`
- `stm_unresolved`

Return Values

Although the return value of this function is of type `INTEGER`, the Documentor enables you to reference this value by name. The name is defined internally as a predefined constant in DGL. The possible values are:

- stm_ac_self_termination
- stm_ac_controlled_termination
- stm_ac_missing

Example

To determine the termination type of the activity `A1` and if the activity is self-terminated write the activity's name, use the following statements:

```
VARIABLE
    ACTIVITY       act_id;
    INTEGER        act_term_type;
    INTEGER        status;
                     .
                     .
                     .
act_id := stm_r_ac ('A1', status);
act_term_type:=stm_r_ac_termination (act_id, status);
IF act_term_type = stm_ac_self_termination THEN
  WRITE ('\n Self-terminated activity:', 'A1');
END IF;
                     .
                     .
                     .
```

stm_r_xx_truth_table

Returns the elements that are implemented as truth tables.

Function type

```
LIST OF STRING
```

For elements

action	an
activity	ac
subroutine	sb

Syntax

```
stm_r_xx_truth_table (xx_id, status)
```

Arguments

Argument	Input/Output	Type	Description
xx_id	In	Statemate element	The element ID
status	Out	INTEGER	The function status code

Status Codes

- stm_success
- stm_error_in_file
- stm_id_out_of_range
- stm_illegal_parameter
- stm_id_not_found
- stm_file_not_found
- stm_missing_name
- stm_missing_field

stm_r_xx_truth_table_expression

Returns the truth table expression for all named elements.

Function type

`STRING`

For elements

action	an
activity	ac
subroutine	sb

Syntax

```
stm_r_xx_truth_table_expression (xx_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`xx_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_error_in_file`
- `stm_id_out_of_range`
- `stm_illegal_parameter`
- `stm_id_not_found`
- `stm_file_not_found`
- `stm_missing_name`
- `stm_missing_field`

stm_r_sb_truth_table_local_data

Returns the list of local data elements defined in the truth table related to the input subroutine.

Function type

```
LIST OF LOCAL_DATA
```

For elements

action	an
activity	ac
subroutine	sb

Syntax

```
stm_r_xx_truth_table_local_data (sb_id, status)
```

Arguments

Argument	Input/Output	Type	Description
sb_id	In	Statemate element	The element ID
status	Out	INTEGER	The function status code

Status Codes

- stm_success
- stm_error_in_file
- stm_id_out_of_range
- stm_illegal_parameter
- stm_id_not_found
- stm_file_not_found
- stm_missing_name
- stm_missing_field

stm_r_xx_type

Retrieves element subtypes for the specified element. Most Statemate elements are divided into classes, referred to as *subtypes*. For example, a state might belong to one of a number of subtypes, such as *and*, *or*, *basic*, *diagram*, *instance*, or *reference*.

You can call this function without indicating the specific element type, as follows:

```
stm_r_type (id, status)
```

Function type

```
INTEGER (predefined constant)
```

For elements

a-flow-line (basic)	ba
a-flow-line (compound)	af
action	an
activity	ac
actor	actor
boundary box	bb
chart	ch
condition	co
connector	cn
data-item	di
data-store	ds
event	ev
field	fd
function	fn
information-flow	if
lifeline	ll
module	md
module-occurrence	om
router	router
state	st
use case	uc
user-defined type	dt

Syntax

```
stm_r_xx_type (xx_id, status)
```

Arguments

Argument	Input/Output	Type	Description
xx_id	In	Statemate element	The element ID
status	Out	INTEGER	The function status code

Status Codes

- stm_success
- stm_id_out_of_range
- stm_id_not_found

Return Values

Although the return value of this function is of type INTEGER, the Documentor enables you to refrence this value by name. The name is internally defined as a predefined constant in DGL. The following table lists the possible values:

Element Type	Function Type	Element Subtype
a-flow-line	stm_a_flow_line_type	stm_af_control
		stm_af_data
action	stm_action_type	stm_an_compound
		stm_an_reference
activity	stm_activity_type	stm_ac_control
		stm_ac_control_instance
		stm_ac_diagram
		stm_ac_external
		stm_ac_instance
		stm_ac_internal
		stm_ac_reference
chart	stm_chart_type	stm_ch_activity
		stm_ch_module
		stm_ch_reference _activity
		stm_ch_reference_module
		stm_ch_reference_state
		stm_ch_state
condition	stm_condition_type	stm_co_compound
		stm_co_primitive
		stm_co_reference

Element Type	Function Type	Element Subtype
connector	stm_connector_type	stm_cn_composition
		stm_cn_condition
		stm_cn_control
		stm_cn_deep_history
		stm_cn_default
		stm_cn_diagram
		stm_cn_history
		stm_cn_joint
		stm_cn_junction
		stm_cn_selection
		stm_cn_termination
data-item	stm_data_item_type	stm_di_compound
		stm_di_alias
		stm_di_constant
		stm_di_primitive
		stm_di_reference
data-store	stm_data_store_type	stm_ds_internal
		stm_ds_reference
event	stm_event_type	stm_ev_compound
		stm_ev_primitive
		stm_ev_reference
field	stm_field_type	stm_fd_primitive
information-flow	stm_information_flow_type	stm_if_explicit
		stm_if_reference
module	stm_module_type	stm_md_diagram
		stm_md_subsystem
		stm_md_environment
		stm_md_reference
		stm_md_instance
		stm_md_storage_module
router	stm_router_type	stm_router_external
		stm_router_internal

Element Type	Function Type	Element Subtype
state	stm_state_type	stm_st_diagram
		stm_st_and
		stm_st_or
		stm_st_instance
		stm_st_reference
		stm_st_basic
subroutine	stm_subroutine_type	stm_sb_reference
user-defined type		stm_dt_primitive
		stm_dt_reference

Example

To retrieve the type of state READY and execute some statements if the state is an or state, use the following statements:

```
VARIABLE
    STATE            st_id;
    INTEGER          st_type;
    INTEGER          status;
                     .
                     .
                     .
st_id := stm_r_st ('READY', status);
st_type := stm_st_type (st_id, status);
IF st_type = stm_st_or THEN
                     .
                     .
                     .
```

If READY is an or-state, the statements following THEN are executed.

stm_r_xx_type_expression

Returns the type expression for the specified element. The expression is the same as used in the properties, reports, and Info.

You can call this function without indicating the specific type, as follows:

```
stm_r_type_expression (id, status)
```

Function type

`STRING`

For elements

condition	co
data-item	di
event	ev
field	fd
local data	ld
subroutine parameter	sp
user-defined type	dt

Syntax

```
stm_r_xx_type_expression (xx_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`xx_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_id_not_found`
- `stm_unresolved`

stm_r_xx_uniquename

Returns the unique path name for the specified element. The name returned by the function contains the minimum number of levels necessary to uniquely identify an element in its chart. It is especially relevant to boxes.

You can call this function without indicating the specific element type, as follows:

```
stm_r_uniquename (id, status)
```

Function type

```
STRING
```

For elements

action	an
activity	ac
actor	actor
boundary box	bb
condition	co
data-item	di
data-store	ds
event	ev
field	fd
function	fn
information-flow	if
lifeline	ll
local data	ld
module	md
router	router
state	st
subroutine	sb
subroutine parameter	sp
use case	uc
user-defined type	dt

Syntax

```
stm_r_xx_uniquename (xx_id, status)
```

Arguments

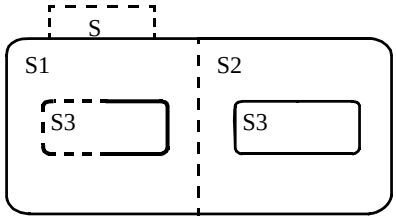
Argument	Input/Output	Type	Description
`xx_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_id_not_found`
- `stm_missing_synonym`

Example

Consider the following statechart:



To retrieve the unique name of the highlighted element, use the following statements:

```
VARIABLE
    STATE           state_id;
    INTEGER         status;
                    .
                    .
                    .
state_id := stm_r_st ('S1.S3', status);
WRITE ('\n Unique Name:',  (stm_r_st_uniquename (
    state_id, status));
                    .
                    .
                    .
```

The state name printed is `S1.S3` (not `S.S1.S3 or S3`). In this example, a unique state name is provided, and this value is used to retrieve the same unique state name from the database. This example demonstrates the value returned by this function, in contrast to the value returned by the function `stm_r_xx_name`.

stm_r_ch_usage_type

Returns the usage type for a chart.

Function type: `INTEGER (predefined constant)`

For elements

chart	ch

Syntax

```
stm_r_ch_usage_type (ch_id, status)
```

Arguments

Argument	Input/ Output	Type	Description
`ch_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_id_not_found`

Return Value

Although the return value of this function is of type `INTEGER`, the Documentor enables you to reference this value by name. The name is defined internally as a predefined constant in DGL. The possible values are as follows:

- `stm_ch_usage_generic`
- `stm_ch_usage_normal`
- `stm_ch_usage_ref_generic`
- `stm_ch_usage_ref_offpage`
- `stm_ch_usage_ref_describing`

stm_r_xx_user_type

Returns the user-defined type ID referenced by the element.

You can call this function without indicating the specific element type, as follows:

```
stm_r_user_type (id, status)
```

Function type: `DATA_TYPE`

For elements

data-item	di
field	fd
local data	ld
subroutine parameter	sp
user-defined type	dt

Syntax

```
stm_r_xx_user_type (xx_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`xx_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_id_not_found`
- `stm_missing_user_type`

stm_r_xx_user_type_name_type

Returns the name type of the user-defined type referenced by the element.

You can call this function without indicating the specific element type, as follows:

```
stm_r_user_type_name_type (id, status)
```

Function type: `INTEGER`

For elements

data-item	di
field	fd
local data	ld
subroutine parameter	sp
user-defined type	dt

Syntax

```
stm_r_xx_user_type_name_type (xx_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`xx_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_ntc_name`
- `stm_ntc_synonym`
- `stm_ntc_unknown`

stm_r_ch_version

Returns the version of the specified chart.

Function type: `STRING`

For elements

chart	ch

Syntax

```
stm_r_ch_version (ch, status)
```

Arguments

Argument	Input/Output	Type	Description
`ch`	In	`CHART`	The chart whose version you want to retrieve
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_not_found`
- `stm_id_out_of_range`
- `stm_unresolved`

stm_r_gds_visibility_mode

Returns the visibility mode for the specified global definition set (GDS).

Function type: `INTEGER`

For elements

element ID	

Syntax

```
stm_r_gds_visibility_mode (gds_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`gds_id`	In	`CHART`	The GDS whose visibility you want to retrieve
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_id_out_of_range`
- `stm_success`

Return Values

Although the return value of this function is of type INTEGER, the Documentor enables you to reference this value by name. The name is internally defined as a predefined constant in DGL. The possible values are as follows:

- `stm_explicit_usage`
- `stm_public_usage`

stm_r_msg_where_tc_begins

Returns the message where the timing constraint begins.

Function type: `MESSAGE`

For elements

message	msg

Syntax

```
stm_r_msg_where_tc_begins (tc_id, status)
```

Arguments

Argument	Input/ Output	Type	Description
`tc_id`	In	`TIMING CONSTRAINT`	The timing constraint
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_not_timing_constraint`

stm_r_msg_where_tc_ends

Returns the message where the timing constraint ends.

Function type: `MESSAGE`

For elements

chart	ch

Syntax

```
stm_r_msg_where_tc_ends (tc_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`tc_id`	In	`TIMING CONTRAINT`	The timing constraint
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_not_timing_constraint`

stm_r_sb_connected_statechart

Returns the ID of the procedural Statechart connected to the specified subroutine.

Function type: `stm_list`

For elements

subroutine	sb

Syntax

```
STM_R_SB_CONNECTED_STATECHART(IN sb: SUBROUTINE, OUT st: INTEGER): CHART;
```

Arguments

Argument	Input/Output	Type	Description
`sb_id`	In	`stm_id`	The element ID.
`status`	Out	`int`	The function status code.

Status Codes

- `stm_success`
- `stm_id_not_found`
- `stm_id_out_of_range`
- `stm_no_connected_chart`

stm_r_sb_connected_flowchart

Returns the ID of the Flowchart connected to the specified subroutine.

Function type:

stm_list

For elements

subroutine	sb

Syntax

```
STM_R_SB_CONNECTED_FLOWCHART(IN sb: SUBROUTINE, OUT st: INTEGER): CHART;
```

Arguments

Argument	Input/Output	Type	Description
`sb_id`	In	`stm_id`	The element ID.
`status`	Out	`int`	The function status code.

Status Codes

- `stm_success`
- `stm_id_not_found`
- `stm_id_out_of_range`
- `stm_no_connected_chart`

stm_r_sb_proc_fch_local_data

Retrieves the local data of the procedural flowchart implemented by the specified subroutine.

Function type: `LIST OF LOCAL_DATA`

For elements

subroutine	sb

Syntax

```
stm_r_sb_proc_fch_local_data (sb_id, status)
```

Arguments

Argument	Input/Output	Type	Description
`sb_id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_missing_local_data`
- `stm_no_connected_chart`

stm_r_xx_des_attr_val

Retrieves design-attribute values associated with a particular design-attribute name for the specified element.

You can call this function without indicating the specific element type, as follows:

```
stm_r_des_attr_val (id, attr_name, status)
```

Function type: `LIST OF STRING`

For elements

activity	ac
action	an
chart	ch
condition	co
data-item	di
data-type	dt
event	ev
field	fd
information-flow	if
local data	ld
state	st
subroutine	sb
subroutine parameter	sp
transitions	tr

Syntax

```
stm_r_xx_des_attr_val(xx_id, des_attr_name, status)
```

Arguments

Argument	Input/ Output	Type	Description
`xx_id`	In	Statemate element	The element ID.
`attr_name`	In	`STRING`	The design-attribute name. The design-attribute name is not case-sensitive.
`status`	Out	`INTEGER`	The function status code. If `attr_name` does not exist for the specified element, `status` receives the value `stm_attribute_name_not_found`.

Status Codes

- `stm_success`
- `stm_des_attribute_name_not_found`
- `stm_id_not_found`
- `stm_id_out_of_range`
- `stm_illegal_name`
- `stm_unresolved`

stm_r_xx_des_attr_name

Returns the names of design-attributes associated with the specified element. Design-attributes are associated with elements via element forms.

You can call this function without indicating the specific type, as follows:

```
stm_r_des_attr_name(id, status)
```

Function type: `LIST OF STRING`

For elements

activity	ac
action	an
chart	ch
condition	co
data-item	di
data-type	dt
event	ev
field	fd
information-flow	if
local data	ld
state	st
subroutine	sb
subroutine parameter	sp
transitions	tr

Syntax

```
stm_r_xx_des_attr_name(xx_id, status)
```

Arguments

Argument	Input/ Output	Type	Description
`xx_id`	In	Statemate element	The element ID.
`attr_name`	In	`STRING`	The design-attribute name. The design-attribute name is not case-sensitive.
`status`	Out	`INTEGER`	The function status code. If `attr_name` does not exist for the specified element, `status` receives the value `stm_attribute_name_not_found`.

Status Codes

- `stm_success`
- `stm_des_attribute_name_not_found`
- `stm_id_not_found`
- `stm_id_out_of_range`
- `stm_illegal_name`
- `stm_unresolved`

stm_r_tt_cell_hyper

Retrieves the contents of the specified cell in the given truth table, including hyperlinks to referenced elements.

Function type: `STRING`

For elements

action	an
activity	ac
subroutine	sb

Syntax

```
STM_R_TT_CELL_HYPER(IN el: ELEMENT ,IN row_num: INTEGER,IN col_num:
INTEGER ,IN format: STRING, OUT status:INTEGER): STRING;
```

Arguments

Argument	Input/Output	Type	Description
`el`	In	Statemate element	The element ID
`row_num`	In	`INTEGER`	The row number of the cell
`col_num`		`INTEGER`	The column number of the cell
`formtat`	In	`STRING`	Either FrameMaker or Word
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_id_not_found`
- `stm_unresolved`
- `stm_missing_truth_table`
- `stm_truth_table_invalid_row`
- `stm_truth_table_invalid_column`

stm_r_tt_row_hyper

Returns a list of strings that represents a row in the truth table, including hyperlinks to referenced elements. Each string in the list includes the text in the truth table cell. The row's index range is [0..num_of_rows-1]. Row 0 returns the list of table header strings.

Function type: `LIST OF STRING`

For elements

action	an
activity	ac
subroutine	sb

Syntax

```
STM_R_TT_ROW_HYPER(IN el: ELEMENT ,IN row_num: INTEGER ,IN format: STRING,
OUT status: INTEGER): LIST OF STRING;
```

Arguments

Argument	Input/Output	Type	Description
`el`	In	Statemate element	The element ID
`row_num`	In	`INTEGER`	The row number of the cell
`formtat`	In	`STRING`	Either FrameMaker or Word
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_id_not_found`
- `stm_unresolved`
- `stm_missing_truth_table`
- `stm_truth_table_invalid_row`
- `stm_truth_table_invalid_column`

stm_r_xx_default_val

Returns the default-value associated with the specified element.

Function type: `STRING`

For elements

condition	co
data-item	di
data-type	dt
local data	ld
field	fd

Syntax

```
stm_r_xx_default_val(id, status)
```

Arguments

Argument	Input/Output	Type	Description
`id`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_id_not_found`

stm_r_component_param_binding

Returns the value bound to the formal parameter of the specified component's instance.

Function type: `STRING`

For elements

activity	ac
condition	co
data-item	di
event	ev

Syntax

```
STM_R_COMPONENT_PARAM_BINDING(IN ins: ELEMENT, IN formal:STRING, OUT status:
INTEGER):STRING;
```

Arguments

Argument	Input/ Output	Type	Description
`inst_boxid`	In	Statemate element	The element ID.
`formal_param_name`	In	`STRING`	The formal parameter name. If this is a data-element (from the information stub matrix in the DDE), the function returns the corresponding data-element. If this argument is the stub's name, the function returns the information flowing on the arrow connected to that stub.
`status`	Out	`int`	The function status code.

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_name_not_found`

stm_r_component_param_mode

Returns the mode of the formal parameter of the specified component's instance.

Function type: `INTEGER`

For elements

activity	ac
condition	co
data-item	di
event	ev

Syntax

```
STM_R_COMPONENT_PARAM_MODE(IN ins: ELEMENT, IN formal:STRING, OUT status:
INTEGER):STRING;
```

Arguments

Argument	Input/ Output	Type	Description
`inst_boxid`	In	Statemate element	The element ID.
`formal_param_name`	In	`STRING`	The formal parameter name. If this is a data-element (from the information stub matrix in the DDE), the function returns the corresponding data-element. If this argument is the stub's name, the function returns the information flowing on the arrow connected to that stub.
`status`	Out	`int`	The function status code.

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_name_not_found`

stm_r_stubs_names

Returns a list of stub names of the specified component instance.

Function type: `LIST OF STRING`

For elements

activity	ac

Syntax

```
STM_R_STUBS_NAMES(IN ins: ELEMENT, OUT status: INTEGER):LIST OF STRING;
```

Arguments

Argument	Input/ Output	Type	Description
`inst_boxid`	In	Statemate element	The element ID
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_name_not_found`

stm_r_information_stub_names

Returns a list of stub names flowing through an info-flow of the specified component instance.

Function type: `LIST OF STRING`

For elements

activity	ac

Syntax

```
STM_R_INFORMATION_STUB_NAMES((IN ins: ELEMENT, IN formal: STRING, OUT status:
INTEGER):LIST OF STRING;
```

Arguments

Argument	Input/ Output	Type	Description
`inst_boxid`	In	Statemate element	The element ID
`formal_param_name`	In	`STRING`	The formal parameter name
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_id_out_of_range`
- `stm_name_not_found`

stm_r_sb_connected_statechart

Returns the ID of the procedural statechart connected to the specified subroutine.

Function type

CHART

For elements

subroutine	sb

Syntax

```
stm_r_sb_connected_statechart (sb_id, status)
```

Arguments

Argument	Input/Output	Type	Description
sb_id	In	Statemate element	The element ID
status	Out	INTEGER	The function status code

Status Codes

- stm_success
- stm_id_not_found
- stm_id_out_of_range
- stm_no_connected_chart

stm_r_sb_connected_flowchart

Function type: CHART

For elements

subroutine	sb

Description:

Returns the ID of the procedural flowchart connected to the specified subroutine.

Syntax

```
stm_r_sb_connected_flowchart (sb_id, status)
```

Arguments

Argument	Input/Output	Type	Description
sb_id	In	Statemate element	The element ID
status	Out	INTEGER	The function status code

Status Codes

- stm_success
- stm_id_not_found
- stm_id_out_of_range
- stm_no_connected_chart

DGL Statement Reference

Document Templates provides general information about the Document Generator Language (DGL), including its overall use, conventions, and a detailed description of the types of statements used in programming document templates. This section provides a comprehensive reference for the DGL statements.

This section presents the statements in alphabetical order by name or keyword. For each DGL statement, the following information is provided:

- **Name** - The keywords used to write the statement.
- **Description** - A detailed description of the statement and what it does.
- **Syntax** - A general syntax description of the statement's structure, including parameters, qualifiers, and options.

The syntax section uses the following conventions:

- Except for literal strings, DGL statements are *not* case-sensitive, and your input can be in either upper- or lowercase. However, upper- and lowercase letter are used in the syntax section to indicate the following:
  - Keywords are shown using uppercase letters.
  - Words written entirely in lowercase indicate variable qualifiers or parameters.
- Items are optional when written inside square brackets [].
- The pipeline symbol (|) denotes "or" between items.
- Ellipses (...) indicate multiple item occurrences. Ellipses can appear horizontally or vertically.

Consider the following examples:

```
ALPHA beta [gamma]
```

`ALPHA` is a statement that takes a required qualifier `beta`, and might take an optional qualifier, `gamma`.

```
JUNK filename [,filename...]
```

`JUNK` is a statement that takes a file name as a parameter. You can specify multiple file names, separating them with commas.

Note

Each DGL statement *must* be concluded by a semicolon. This is shown in the statement syntax. Flow control and structure statements can contain a number of statements. Multiple statements can be contained on a single line, or a single statement can span several lines.

- **Notes** - Special points of interest regarding the use of the statement.
- **Examples -** Examples that illustrate how to use the statement.
- **See Also** - This refers you to the section of this manual where the particular statement is explained.

The remainder of this section presents the DGL statements in alphabetical order.

- SELECT/WHEN
- STOP
- TABLE
- TEMPLATE
- VARIABLE
- VERBATIM
- WHILE/LOOP
- WRITE

ASSIGNMENT

Description

Assigns the value of an expression to a variable. An assignment statement should be interpreted as follows:

The variable on the left-hand side of the statement is assigned the value of the expression on the right-hand side.

The expression and the variable must be of the same or compatible type. If the expression is of type `STATE`, `ACTIVITY`, and so on, the variable is of the same type, or of type `ELEMENT`.

Syntax

```
variable:= expression;
```

Notes

The expression and the variable must be of the same or compatible type. If the expression is of type `STATE`, `ACTIVITY`, and so on, the variable is of the same type or of type `ELEMENT`.

Example

```
VARIABLE
      LIST OF STATE      st_list;
      LIST OF ACTIVITY   act_list;
      LIST OF ELEMENT    el_list;
BEGIN
             .
             .
   el_list  := st_list + act_list;
             .
             .
END:
```

See Also

- **CONSTANT**
- **PARAMETER**
- **VARIABLE**
- **TEMPLATE Statement**

BEGIN

Description

Delineates the boundaries of a template section. Any number of other DGL statements can be between the `BEGIN` and `END`. The statements are optional; a section having no statements between the `BEGIN` and the `END` is legal.

Syntax

```
BEGIN
[statement;]
    .
    .
    .
END;
```

Parameters

Parameter	Description
`statement`	A comment, or DGL statement other than structure or declaration type statements

Notes

`BEGIN` and `END` always appear as a pair. Note that there is no end-of-statement symbol (;) after `BEGIN`.

Example

```
SEGMENT SEG1;
 VARIABLE
   STRING name;
   STATE state-chart;
   INTEGER status;
 BEGIN
   name:= stm_r_st_name (state_chart, status);
   IF status = stm_success AND name <> '' THEN
     WRITE ('.sh 1"' , name, '"\n');
     WRITE ('\n');
   ELSE
     WRITE ('empty name or error status\n');
   END IF;
 END ;
```

See Also

- COMMENT
- END
- PROCEDURE
- SEGMENT
- TEMPLATE
- Structure Statements

CLOSE

Description

Closes files after they have been opened with the `OPEN` statement. After opening a file and passing information to and from it, use the `CLOSE` statement to prevent further information from being passed to or from it.

Syntax

```
CLOSE (fl);
```

Parameters

Parameter	Description
`f1`	An identifier of type `FILE`, previously assigned by the `OPEN` statement

See Also

- **OPEN**
- **READ**
- **VARIABLE Statement**

COMMENT

Description

Prefaces a comment in a template.

Comments are used in templates for documentation purposes. When processing a template, the Documentor compiler ignores these comments. A comment can contain any printable ASCII character. The comment symbol (double-dash) can appear anywhere a space can, except within a literal string.

Comments are preceded by two dashes. The comment symbol can start anywhere in a line, except within a literal string. The Documentor ignores all the characters after the comment symbol until the end of the line.

For example:

```
-- This line contains program comments.
```

Syntax

```
-- free-text
```

Parameters

Parameter	Description
`free-text`	Any printable string

Example

```
-- This is a valid comment line.
a:=a+1; -- This is another comment.
```

See Also

- **BEGIN**
- **END**
- **PROCEDURE**
- **SEGMENT**
- **TEMPLATE**
- **Structure Statements**

CONSTANT

Description

Declares constants (identifiers whose values do not change throughout the template or during execution).

The keyword `CONSTANT` appears only once in the declaration section, before the data-type assignments for constants. Each data-type statement can be followed by as many identifiers of the same type as you want to define. For example:

```
CONSTANT integer a:=1, b:=2, c:=3;
```

Similarly, as many type statements as you want to define can follow the `CONSTANT` keyword. For example:

```
CONSTANT
   STRING activity_name := 'Print';
   FLOAT a:=3.243;
   INTEGER c:=6;
```

Note: The constant type cannot be a Statemate element or a `LIST OF` type.

The identifiers in the `CONSTANT` statement must have their values assigned in the statement. The value can be any expression that does not contain variables or parameters.

Syntax

```
CONSTANT
type identifier := value [, identifier := value, ...];
[ type identifier := value, ...;]
```

Parameters

Parameter	Description
`type`	The type of constant (integer, float, string, or Boolean)
`identifier`	The name of the constant
`value`	An expression that specifies the value of the constant

Example

```
CONSTANT
STRING  out_file :=’/usr/group/doc12.stm’ ;
STRING  in_file := ’/usr/group/insert12.doc’;
INTEGER  page_length := 66, page_width:=72;
```

See Also

- **ASSIGNMENT**
- **PARAMETER**
- **VARIABLE**
- **TEMPLATE Statement**

END

Description

Used as part of the `BEGIN/END` construct in the delineating of template sections.

See the **BEGIN** statement for more information.

Syntax

```
BEGIN
[statement;]
    .
    .
    .
END;
```

See Also

- **BEGIN**
- **COMMENT**
- **PROCEDURE**
- **SEGMENT**
- **TEMPLATE**
- **Structure Statements**

EXECUTE

Description

Invokes a program external to the Statement system. This statement invokes a program that is external to Statemate. The Documentor searches for the program name using the regular system search path, invokes the program, and sends its standard output to the document's output segment file.

The `EXECUTE` statement function operates as a function that returns either `stm_success` or `stm_error`.

Syntax

```
[status := ] EXECUTE (calling_sequence);
```

Parameters

Parameter	Description
`calling_sequence`	A string expression (or string literal) that specifies the name and parameters of an external program. The `calling_sequence` is written in the same manner as when invoking the program from the shell command line. Therefore, the calling sequence can include arguments to be passed to the program. However, no evaluation of these arguments is done in the template—the calling sequence is passed as a pure string.

Example

This following statement calls the operating system function `DATE` and writes the date to the output file:

```
EXECUTE ('date');
```

See Also

- **INCLUDE**
- **REPORT**
- **TABLE**
- **VERBATIM**
- **WRITE**
- **READ Statement**

EXIT

Description

Exits from the current loop to the statement after the loop construct or to the construct that contains the current loop.

Typically, a condition would be tested in a loop, and the exit would be based upon the evaluation of that condition.

Syntax

```
EXIT;
```

Note

`EXIT` can be used only inside a `FOR` and `WHILE` loop.

Example

This example continues the execution of the statements between the `LOOP` and `END LOOP`, depending on the value of the condition `a > b`.

The iterations go on as long as the status `st` is equal to 0. When `st` is not equal to `stm_success`, it causes the iterations to stop. The `IF` statement here is used to force an abnormal `EXIT` from within the `WHILE` loop.

The execution resumes at the next statement after the `END LOOP`.

```
WHILE a > b LOOP
   md_id := stm_r_md (name, st);
   IF st <> stm_success THEN
      WRITE ('Illegal Status');
      EXIT;
   END IF;
      .
      .
   END LOOP;
```

See Also

- **FOR/LOOP**
- **IF/THEN/ELSE**
- **SELECT/WHEN**
- **STOP**
- **WHILE/LOOP**
- **Control Flow Statements**

FOR/LOOP

Description

Provides iterative execution of template statements.

The `FOR/LOOP` construct is used to execute iterative DGL statements. This statement executes the statements between `LOOP` and `END LOOP` for each item in the specified list. The identifier is a variable whose value is set sequentially to the items in the list. This variable can be used within the body of the loop, but its value cannot be reassigned.

Alternatively, a range of integers can be specified in place of the `list`, as in the following example:

```
FOR i IN {1..100} LOOP
   statement;
      .
      .
      .
END LOOP;
```

Syntax

```
FOR identifier IN list
LOOP [statement;]
    .
    .
    .
END LOOP;
```

Parameters

Parameter	Description
`identifier`	A variable whose value is set sequentially to the items in the list. The type of the `identifier` must match the type of the `list`. This variable can be used within the body of the loop.
`list`	A list expression.
`statement`	A DGL statement.

Notes

The type of the variable identifier must match the type of the list. The list can be a range of integers written as follows, where *x* is an integer variable:

```
FOR x IN {1..100} LOOP ...
```

Example

This example writes to the output file the names of all the states in the list `statelist`:

```
FOR id IN statelist LOOP
   WRITE ('\n', stm_r_st_name (id, status));
END LOOP;
```

See Also

- **EXIT**
- **IF/THEN/ELSE**
- **SELECT/WHEN**
- **STOP**
- **WHILE/LOOP**
- **Control Flow Statements**

IF/THEN/ELSE

Description

Provides conditional execution of template statements.

The `IF/THEN/ELSE` construct is used for conditional execution of DGL statements. The statements following the `THEN` (and before any `ELSE`) are executed if `boolean` evaluates to `TRUE`. If it evaluates to `FALSE`, the statements following `ELSE` are executed, when present.

Syntax

```
IF boolean THEN
   [statement;]
          .
          .
          .
   [ELSE
          [statement;]
                 .
                 .
                 .
   ]
END IF;
```

Parameters

Parameter	Description
`boolean`	A Boolean expression
`statement`	A DGL statement

Example

```
IF a >= b THEN
     EXECUTE (’DATE’);
     INCLUDE (’sample.txt’);
  ELSE
     WRITE (’a is less than b’);
END IF;
```

See Also

- **EXIT**
- **FOR/LOOP**
- **SELECT/WHEN**
- **STOP**
- **WHILE/LOOP**
- **Control Flow Statements**

INCLUDE

Description

Copies the text of the specified file to an output segment file. The text is passed to the file verbatim and can contain formatting commands for the format processor used to produce the formatted document.

The file can be an include file within Statemate or a file outside Statemate in the general file system of the host computer.

Syntax

```
INCLUDE (file_description [, status])
```

Parameters

Parameter	Description
`file_description`	A file name or an identifier of type `FILE`. When it is a file name, it can be any string expression, such as a literal string inside quotes (for example, `'ABC'`) or an evaluated expression that produces a file name (for example, a variable that contains a file name). The file name can include the directory path name of the file, using the file name conventions of the host operating system. If you do not specify the path name, the Documentor searches the workarea for the file. When the `file_description` is a file identifier, the file from which you are copying text must first be opened with an `OPEN` statement in `OUTPUT` mode (see **OPEN**).
`status`	The function status code. It returns one of the following values: • `stm_success`—Successful execution of the statement. • `stm_error`—Failure.

Example

In this example, a UNIX file external to Statemate is included in the output segment:

```
INCLUDE('/usr/group/gx_file');
```

See Also

- **EXECUTE**
- **REPORT**
- **TABLE**
- **VERBATIM**
- **WRITE**
- **READ Statement**

OPEN

Description

Opens files or the dialog area so text can be written to or read from them.

This statement is used with `mode=OUTPUT` to open a file or the dialog area so that a subsequent `WRITE` statement can pass text to it, and with `mode=INPUT` for a subsequent `READ` statements.

```
When writing to the dialog area, the syntax is as follows:
   OPEN (fl, 'DIALOG', OUTPUT);
```

Syntax

```
OPEN (fl, file_name, mode [, status]);
```

Parameters

Parameter	Description
`f1`	An identifier of type `FILE`, assigned by the `OPEN` statement.
`file_name`	The name of the file to (or from) which to pass the text, or the dialog area. If the file opened for output does not exist, this statement creates a new file. If the file exists, the `OPEN` statement overwrites the existing contents of the file. To open the dialog area, use the string `'DIALOG'` (with apostrophes) for this parameter. Open the dialog area so you can pass run-time messages to it.
`mode`	If `mode=OUTPUT`, the `OPEN` statement opens a file or the dialog area so a subsequent `WRITE` statement can pass text to it. If `mode=INPUT`, the `OPEN` statement opens a file or the dialog area so a subsequent `READ` statement can pass text to it.
`status`	The function status code. It returns one of the following values: • `stm_success`—Successful execution of the statement. • `stm_error`—Failure.

Notes

The name of the file is specified in accordance with the conventions of the operating system. If the directory is not specified as part of the file name, the workarea is searched for the file. If the file opened for output does not exist, this statement creates a new file. If the file exists, it is initialized by the `OPEN` statement (the written information overwrites the existing contents of the file).

See Also

- **CLOSE**
- **READ**
- **VARIABLE Statement**

PARAMETER

Description

Declares variables and values within a template.

There are *template parameters* for the entire template and *procedure parameters* for procedures. Template parameters are variables whose value can be changed interactively when the template is executed.

You can declare template parameters only in the initiation section. The keyword `PARAMETER` appears only once in the declaration section, before the data-type assignments for parameters. Each data-type statement can be followed by as many identifiers of the same type as you want to define. For example:

```
PARAMETER STRING activity_name, state_name, event_name;
```

As many type statements as desired can follow the `PARAMETER` keyword. For example:

```
PARAMETER
   STRING activity_name;
   FLOAT a:=3.243;
```

Template parameters cannot hold a Statemate element, although it is legal for a procedure parameter to be of this type. Procedure parameters are In/Out parameters.

Value assignments for parameter statements are optional and are allowed only for template parameters. If it is assigned, it represents the default value of the parameter at the first generation of a particular document. The value can only be a literal constant, *not* a constant identifier or an expression.

> **Note:** Avoid changing template parameters within the template. In addition to being confusing, there might be inconsistent results when parameters are changed within segments.

Syntax

```
PARAMETER
type identifier [:= value][, identifier [:= value],...];
[type identifier [:= value],...]
.
.
.
```

Parameters

Parameter	Description
`type`	The parameter type (integer, float, string, or Boolean—or a list of these types).
`identifier`	The name of the parameter. The maximum length of the identifier is 16 characters. If you specify more than 16 characters, the name is truncated.
`value`	value

Example

```
PARAMETER
    STRING statechart_name;
    INTEGER page_width := 80;
```

See Also

- **ASSIGNMENT**
- **CONSTANT**
- **VARIABLE**
- **TEMPLATE Statement**

PROCEDURE

Description

Begins a procedure section. It assigns an identifying name to the procedure and defines the return type, if the function returns a value.

The definition of a procedure is as follows:

```
PROC <procname> [return <type>];
[PARAMETER
 <type> <parm1>;
....
 <type> <parmN>;
]
[VARIABLE
  var section;
]
BEGIN
  <statements>;
  return <expression>;
END;
```

Syntax

```
PROCEDURE procedure_name [RETURN type];
```

Parameters

Parameter	Description
`procedure_name`	The name of the procedure. The maximum length is 16 characters.
`RETURN type`	Assigns the value that is returned by the function and returns from the procedure.

Notes

The scoping rules for procedures are as follows:

- For simplification reasons, procedures are not lexically nested within other procedures, nor are they nested within segments.
- Like segments, procedures can see template global variables.
- Procedures can be referenced by any template segment and can be defined anywhere within the template (but not within segments).
- Parameters are always In/Out.

Example

```
TEMPLATE sss;
PARAMETER
   string     plot_dir         := '';
   string     appendix         := '';
VARIABLE
   string     module_name;
   string      key_bg , key_en;

BEGIN
   open (da, 'DIALOG', OUTPUT);
   .....
END;


PROCEDURE get_generic_info;
PARAMETER
 activity  cap;      -- Input
 activity  capdef;
 boolean   is_generic;
 element   chrt;      -- Outputs
VARIABLE
 integer   st, st1, st2, st3;
 activity  acttmp;
BEGIN
  capdef := cap;
  is_generic := false;

  IF (stm_r_mx_generic_instance_mx ({cap}, st) <> {})
     THEN
     chrt:=stm_list_first_element
          (stm_r_ch_defining_ac({cap},st),st2);
   IF (st = stm_success AND st2 = stm_success) THEN
            acttmp := stm_list_first_element
            (stm_r_ac_root_in_ch({chrt},st),st2);
    IF (st = stm_success AND st2 = stm_success) THEN
            acttmp := stm_list_first_element
            (stm_r_ac_internal_ac(
            stm_r_ac_logical_sub_of_ac ({acttmp}, st)
            ,st2),st3);
     IF (st = stm_success AND st2 = stm_success AND
              st3 = st2) THEN capdef := acttmp;
              is_generic := true;
     END IF;
    END IF;
   END IF;
  END IF;
END;


      PROCEDURE put_plot_lines;
      PARAMETER
       float acty; -- Whatever it means
       string title;
      VARIABLE
       integer lines;
      BEGIN
           acty := (acty/INCH_PER_LINE) + 0.5;
           lines := 0;
           WHILE (acty > 0) LOOP
              acty := acty-1;
```

```
        lines := lines + 1;
     END LOOP;
     write ('\n<HardReturn>');
     write ('\n First line of plot ',title,'
         (Delete between lines)--->>>');
     WHILE (lines > 0) LOOP
        write ('\n<HardReturn>\nReserved for plot ---> ',fl);
          lines := lines - 1;
     END LOOP;
     write ('\n<HardReturn>');
     write ('\n Last line of plot  ',title,'
     (Delete between lines)--->>>');
END;



PROCEDURE put_bindings ;
PARAMETER
 element cap;
VARIABLE
 element par;
 list of element formals;
 integer st, etype;
 string title, stype, actual;
BEGIN
    -- Specify the bindings
    formals := stm_r_mx_parameter_of_ch
              (stm_r_ch_defining_ac ({cap}, st), st);
    .........
END;  -- Put bindings


PROCEDURE put_mini_spec;
PARAMETER
 activity act;
VARIABLE
 integer st1,st;
 string mini_spec;
BEGIN
    mini_spec := stm_r_ac_mini_spec (act, st1);
    IF (st1 = stm_success) THEN
     write ('\n<Bold>\n');
     write ('Mini-Spec :');
     write ('\n<NoBold>\n');
     write ('\n\n');
     lwrite (mini_spec);
     write ('\n\n');
    END IF;
    IF (st <> stm_success and st1 <> stm_success) THEN
      write ('TBD.');
    END IF;
END ; -- put_mini_spec

SEGMENT body;

 cap:=....
 put_bindings (cap);
 put_mini_spec (cap);

.....
END;
```

See Also

- **BEGIN**
- **COMMENT**
- **END**
- **SEGMENT**
- **TABLE**
- **Structure Statements**

READ

Description

Reads a line of information from an external file into variables in the template. The numeric elements in the input line are separated by blanks or tabs. Reading to a string variable reads the rest of the line.

The `READ` statement can operate as a function that returns either `stm_success`, `stm_cannot_read_file, or stm_end_of_file.`

Syntax

```
READ (fl, variable1, variable2, ...);
```

Parameters

Parameter	Description
`f1`	An identifier of type FILE, that points to a file previously opened by the OPEN statement in INPUT mode
`variable1, variable2...`	Identifiers of type integer, float, or string

Example

In this example, `i` is an integer and `str` is a string. This statement, when applied to an input line `12 May 2003`, results in `i=12, str='May 2003'.`

```
READ(fd, i, str)
```

See Also

- **CLOSE**
- **OPEN**
- **VARIABLE Statement**

REPORT

Description

Invokes the Reports tool to generate a predefined report as part of a document.

For example:

```
stm_rpt_tree(list,5);
```

This call produces a tree report for the items in a list represented by the variable `list` to a depth of 5 in the hierarchy. The report is included in the output file.

The output from the Reports tool contains formatting commands applicable to the format processor attached to the template. If no formatter is specified, the Reports tool cannot be invoked and the Documentor generates an error message.

Each of the predefined reports is invoked for a list of elements. The input parameter that represents this list is denoted by a variable name that must be of type `list of` one of the Statemate element types. This variable, along with all other identifier names used in the calling sequence, must be declared in an appropriate declaration section. For example:

```
VARIABLE
  LIST OF ACTIVITY ac_list;
```

The identifier `ac_list` can be assigned a list of activities and then be included in a statement that generates a property report, as follows:

```
stm_rpt_dictionary (ac_list,...);
```

The report is generated for each item in the list represented by the `ac_list` variable.

A number of arguments are used to define the parameters for each report. Some are called “single-character string arguments,” which are used to indicate restricted parameter choices. Consider the following interface report statement

```
stm_rpt_interface (elist, 'A', ...);
```

The value of the second argument, `A`, indicates that the interface report should be of type `activities`; specifying an `M` for this parameter would indicate that the report should be generated for `modules`.

The single-character string arguments can be more than one character, but only the first character of the string is actually passed to the Reports tool. If a non-valid character is passed to the Reports tool, the report is not generated and an error status code is returned.

Some of the arguments are Boolean and are evaluated as `TRUE` or `FALSE` to indicate whether some parameter is set. For example, consider the following property report statement:

```
stm_rpt_dictionary (elist, true, ...);
```

The Boolean constant `true` indicates that the long description will be included in the report.

Syntax

```
stm_rpt_<report_name> (report_parameters);
```

Parameters

Parameter	Description
`report_name`	The type of report to be generated
`report_parameters`	The report parameters

Supported Statemate Reports

The supported report types are as follows:

- **Attribute Report**
- **Property Report**
- **Interface Report**
- **List Report**
- **N2 Chart Report**
- **Protocol Report**
- **Resolution Report**
- **Structure Report**
- **Tree Report**

Attribute Report

Syntax

```
stm_rpt_attribute (elist, attrs, attr_title);
```

Parameters

Parameter	Description
`elist`	A list expression of Statemate elements for which the report is produced.
`attrs`	A list of strings that contains the specific attribute names for which the report should be generated. If this list is empty, the report retrieves all the attributes for each element.
`attr_title`	A string indicating that the attribute value will precede its element name in the report.

Property Report

Syntax

```
stm_rpt_dictionary (elist, ldes, attr, attr_title);
```

Parameters

Parameter	Description
`elist`	A list expression of Statemate elements for which the report is produced
`ldes`	A Boolean expression indicating whether to include the long description of each element in the report
`attr`	A Boolean expression indicating whether to include the attributes of an element in the report
`attr_title`	A string indicating the attribute names whose value will precede the element name in the report

Interface Report

Syntax

```
stm_rpt_interface (elist, rtype, chart, lact lmod, ftype,
   dis, names);
```

Parameters

Parameter	Description
`elist`	A list expression that must be of the type list of modules, for which the report is produced.
`rtype`	A single-character string argument indicating the report type, as follows: • **A**—Activity interface report • **M**—Module interface report • **I**—Information interface report
`chart`	A single-character string argument indicating which arrows are taken into account when the report is generated. The possible values are as follows: • **A**—Activity-chart arrows • **M**—Module-chart arrows
`lact`	An argument of type `list of activities` indicating which activities are taken into account when the report is generated. If `lact` is an empty list, the default is all activities implemented by the center module. If chart is 'M', the lact parameter has no use, but still must be specified. For simplicity, use the null list (null).
`lmod`	An argument of type `list of modules` indicating the side modules that interface with the central module for which the report is to be generated. If `lmod` is empty, the default behavior is to use *all* modules except the center module's own ancestors and descendants.
`ftype`	A single-character string argument indicating the kind of information flow to appear in the report. The possible values are as follows: • **D**—Data-flows • **C**—Control-flows • **B**—Both data-flows and control-flows
`dis`	A single-character string argument indicating the kind of information to appear in the report. The possible values are as follows: • **I**—Flow labels • **P**—Parent information items • **B**—Basic information items
`names`	A single-character string argument specifying how names appear in the report. The possible values are as follows: • **N**—The name appears for elements that flow between the boxes. • **S**—The synonym appears for elements that flow between the boxes.

List Report

Syntax

```
stm_rpt_list (elist);
```

Parameters

Parameter	Description
`elist`	A list expression of Statemate elements for which the report is produced

N2 Chart Report

Syntax

```
stm_rpt_n2chart(elist, names, level, env, chart, dis, ftype);
```

Parameters

Parameter	Description
`elist`	A list expression that must be of the type list of modules or list of activities, which specifies the elements in the diagonal.
`names`	A single-character string argument specifying how names appear in the report. The possible values are as follows: • **N**—Names of the elements appear on the diagonal of the matrix. • **S**—Synonyms of the elements appear on the diagonal of the matrix.
`level`	A single-character string argument indicating what appears on the diagonal when both parent box and sub-box are in the list. The possible values are as follows • **B**—The sub-box is placed on the diagonal of the matrix. • **P**—The parent box is placed on the diagonal of the matrix.
`env`	A Boolean expression. If this is TRUE, the environment is added to the matrix.
`chart`	A single-character string argument indicating which arrows are taken into account when the report is generated. The possible values are as follows: • **A**—Activity-chart arrows • **M**—Module-chart arrows

Parameter	Description
`dis`	A single-character string argument indicating the kind of information to appear in the report. The possible values are as follows: • **I**—Flow labels • **P**—Parent information items • **B**—Basic information items
`ftype`	A single-character string argument indicating the kind of information flow to appear in the report. The possible values are as follows: • **D**—Data-flows • **C**—Control-flows • **B**—Both data-flows and control-flows

Protocol Report

Syntax

```
stm_rpt_protocol (elist, attr_title);
```

Parameters

Parameter	Description
`elist`	A list expression of Statemate elements for which the report is produced.
`attr_title`	A string indicating the attribute name whose value will precede the element name in the report.

Resolution Report

Syntax

```
stm_rpt_resolution (clist, type)
```

Parameters

Parameter	Description
`clist`	A list of charts. It determines the scope of the report. The stm_type of elements to include in the report. The possible types are as follows: • stm_textual • stm_graphical • stm_mixed • stm_state • stm_module • stm_activity • stm_data_store

Structure Report

Syntax

```
stm_rpt_structure (elist, width);
```

Parameters

Parameter	Description
`elist`	A list expression of Statemate elements for which the report is produced.
`width`	An integer argument indicating the page width (in characters) to be used for the report.

Tree Report

Syntax

```
stm_rpt_tree (elist, depth);
```

Parameters

Parameter	Description
`elist`	A list expression of Statemate elements for which the report is produced.
`depth`	An integer argument indicating the to what hierarchical level the report should be generated. For all levels, use the value “99”.

See Also

- **EXECUTE**
- **INCLUDE**
- **TABLE**
- **VERBATIM**
- **WRITE**
- **READ Statement**

SEGMENT

Description

Starts a new segment section. It is the first statement of a segment and assigns an identifying name to it. The segment name is used by the Documentor in its operation forms to identify the output segments see **Creating a Document**).

Syntax

```
SEGMENT segment_name;
```

Parameters

Parameter	Description
`segment_name`	A name used to identify the segment. This identifier is limited to a maximum length of 16 characters. Characters in excess of this amount are truncated and do not appear when the segment names are displayed in the operations forms.

Example

```
SEGMENT chapter_2B;
```

See Also

- **BEGIN**
- **COMMENT**
- **END**
- **PROCEDURE**
- **TEMPLATE**
- **Structure Statements**

SELECT/WHEN

Description

Performs conditional execution of DGL statements. This statement is more powerful than the `IF/THEN/ELSE` statement in that it allows you to systematically list multiple conditions for statement execution.

Note that `WHEN` statements are composed of two parts: the trigger to the left of the arrow, and statements on the right side of the arrow. The trigger is any valid Boolean expression. The statements following a trigger are performed only when the trigger is `TRUE`. Whether or not these statements are actually executed also depends on the value of the `selection_mode`. In addition:

- The `WHEN ANY` statements are executed when one or more of the preceding `WHEN` statements have been executed.
- The `OTHERWISE` statements are executed only if no `WHEN` statement within the `SELECT` construct is triggered.

Regardless of the mode, when all of the triggers are false, the tool will execute the `OTHERWISE` clause, if it exists.

Syntax

```
SELECT [selection_mode]

    WHEN trigger => statements
  [ WHEN trigger => statements ]
            .
            .
  [ WHEN ANY => statements ]
            .
  [ WHEN trigger => statements ]
            .
            .
  [ WHEN ANY => statements ]
            .
            .
  [ OTHERWISE => statements ]

END SELECT ;
```

Parameters

Parameter	Description
`trigger`	A Boolean expression.
`selection_mode`	Specifies the selection mode, which determines the way the statements are checked for possible execution. The possible values are as follows: • **FIRST**—Only the first true trigger in the entire `SELECT` construct is executed; the rest are ignored, regardless of whether their triggers are `TRUE`. This is the default value. • **ANY**—If the selection mode is `ANY`, the statements are executed whenever their corresponding trigger is `TRUE`.
`statement`	The DGL statements to invoke.

Notes

Both the `WHEN ANY` and the `OTHERWISE` statements are optional.

The Documentor permits the nesting of `SELECT/WHEN` constructs.

Example

Consider the following example, where `a`, `b`, and `c` are numeric variables:

```
SELECT ANY
  WHEN a = 5 => b := 10 ;
  WHEN a > b => b := 10 ;
  WHEN a = 0 => b := 0 ;
  WHEN ANY  => write ('a may influence b') ;
  WHEN c = 5 => b := 5 ;
  WHEN c > b => b := 5 ;
  WHEN c = 0 => b := 0 ;
  WHEN ANY  => write ('c may influence b') ;
  OTHERWISE => write ('b has not been changed') ;
END SELECT;
```

The execution is determined by the following:

- Each `WHEN` statement is triggered if its corresponding expression is evaluated to `TRUE`.
- The first `WHEN ANY` statement is triggered if a is equal to 5, greater than `b`, or equal to zero. The second `WHEN ANY` statement is triggered if at least one of the previous conditions is true with respect to the variable `c` instead of `a`.
- The `OTHERWISE` statement is triggered only if the value of `b` has not been changed within the `SELECT` statement's evaluation.

When processing a `WHEN ANY` statement, the Documentor only "looks back" to the previous `WHEN ANY` construct (if one exists). Therefore, in this example, if `a = 0` and

none of the tests of c were true, the WRITE statement's message "c may influence b" is not issued.

Using the same example, what would happen if the selection mode is FIRST instead of ANY, and the conditions a > b and c = 5 are both true?

In this case, the assignment b:= 10 and the first write message (the corresponding WHEN ANY statement) are executed. The assignment of b to 5, along with its corresponding WHEN ANY statement are not done because c = 5 is not the first true trigger.

The statements following the => symbol in the WHEN constructs can be any valid DGL statements. You can even enter a SELECT construct at this point. This allows you to nest SELECT constructs. There is no limit to the depth of nested SELECT blocks.

See Also

- **EXIT**
- **FOR/LOOP**
- **IF/THEN/ELSE**
- **STOP**
- **WHILE/LOOP**
- **Control Flow Statements**

STOP

Description

Stops execution of the template.

Typically, a specific condition is tested and the template is stopped if this condition has a value for which further processing is meaningless.

Syntax

```
stop;
```

Example

The following example checks whether the specified `system_name` is valid. If not (if an error has been detected), a message is issued to the dialog area and the template is stopped.

```
md := stm_r_md (system_name, status);

IF (status <> stm_success) THEN
   WRITE (dialog_area, 'Execution Stopped due
      to error');
STOP;
END IF;
```

See Also

- **EXIT**
- **FOR/LOOP**
- **IF/THEN/ELSE**
- **SELECT/WHEN**
- **WHILE/LOOP**
- **Control Flow Statements**

TABLE

Description

Creates a simple table to be included in your document. You specify the number of columns and their widths, and the information to be included in the table in the statement parameters.

Syntax

```
stm_table_simple (title, columns, contents, page_width, page_height [,
anchor]);
```

Parameters

Parameter	Description
`title`	The title of the table. The title is centered over the table.
`columns`	A list of integers that specifies the width of each column, in number of characters. For example, `{16} & {16} & {20}` specifies a table having three columns, the first two columns being 16 characters wide and the last column being 20 characters wide.
`contents`	A list of strings that contain the information to be entered into each cell of the table. The Documentor fills the table horizontally, row by row, depending on how many columns were specified. For example, if there are three columns, and you specified the following contents: {'Project Name'} & {'Date'} & {'Location'} & {'Alpha'} &{'April 2004'} & {'Boston'} The resultant table would be: **Project Name** | **Date** | **Location** Alpha | April 2008 | Boston
`page_width`	Specifies the width of the page, in inches. This parameter is relevant for Interleaf only—for other systems, specify 0.0 for this parameter.

`page_height`

Specifies the height of the page, in inches. This parameter is relevant for Interleaf only—for other systems, specify 0.0 for this parameter.

`anchor`

This parameter is relevant for Interleaf only. For formatters other than Interleaf, precede the table with your system's "no fill" and "no adjust formatting" commands.

For Interleaf, this parameter is a character string that indicates where to place the table. The possible values are as follows:

`A—At the anchor.`

`F—Following the anchor. This is the default value.`

Example 1

This example produces a table named "Table-1", where the page is 6 inches wide and 9.5 inches long. The column widths are determined by the list held in the integer list variable `col_list`, and the cell contents are held in the string list variable `cell_list`.

```
stm_table_simple ('Table-1', col_list, cell_list,
   6.0, 9.5);
```

Example 2

The following example shows how to generate a table using function calls. Note that in the example, new values are repeatedly assigned to the `List_str` variable to build the table.

The first statement uses the nroff commands for no fill (`.nf`) and no adjust (`.na`). These commands cause the word processor to take the text "as-is." Some word processors see this mode as *verbatim* or *literal*. The last statement uses the nroff commands `.fi` and `.ad` to return to fill and adjust modes.

```
WRITE ('\n.nf \n.na \n');

BEGIN
 List_str:= {'ACTIVITY NAME'} & {'ID'} & {'LANGUAGE'};'
 act_list:=stm_r_ac_logical_desc_of_ac({act_chart},st);
 FOR act IN act_list LOOP
   List_str:=List_str & {stm_r_ac_name (act, st)};

   attr_list:=stm_r_ac_attr_val (act, 'ID_NUMBER', st);

   IF (st = stm_success) THEN
      attr_val:=stm_list_first_element (attr_list, st);
      List_str:=List_str & {attr_val};
   ELSE
       List_str:=List_str & { 'N/A'};
   END IF;

  attr_list:=stm_r_ac_attr_val(act, 'LANGUAGE', st);
   IF (st = stm_success) THEN
      attr_val:=stm_list_first_element (attr_list, st);
```

```
      List_str:=List_str & {attr_val};
   ELSE
        List_str:=List_str & {'N/A'};
   END IF;
 END LOOP;

 WRITE ('\n.nf\n.na\n');
 title := 'Table CC1. Simple Table Example';
 Col_list := {16} & {10} & {40};
 stm_table_simple(title, Col_list, List_str, pg_w, pg_h,
    'A');
 WRITE ('\n.fi\n.ad\n');
END;
```

The formatted output is as follows:

ACTIVITY NAME	ID	LANGUAGE
FUNCTION1	AC1-A1.1	FORTRAN
FUNCTION2	AC1-A1.2	N/A
FUNCTION3	AC1-A1.5	ADA
FUNCTION4	AC1-A1.4	N/A

See Also

- **EXECUTE**
- **INCLUDE**
- **REPORT**
- **VERBATIM**
- **WRITE**
- **READ Statement**

TEMPLATE

Description

Begins a template.

The `TEMPLATE` statement is the first statement in the template. It assigns an identifying name to the template. This name is used for internal documentation purposes only and does not have to correspond to the name you use to designate the template in the Create Template form.

Syntax

```
TEMPLATE template_name;
```

Parameters

Parameter	Description
`template_name`	The name of the template. This name is for internal documentation purposes only.

Example

```
TEMPLATE document_574B;
```

See Also

- **BEGIN**
- **COMMENT**
- **END**
- **PROCEDURE**
- **SEGMENT**
- **Structure Statements**

VARIABLE

Description

Declares variables, which are identifiers whose values can be changed in other DGL statements.

Variable values can be global when included in the initiation section, or local to a segment.

The keyword `VARIABLE` appears only once in the declaration section, before the data-type assignments for variables. Each data-type statement can be followed by as many identifiers of the same type as you want to define. For example:

```
VARIABLE STRING act_name, act_syn, act_desc;
```

Similarly, as many type statements as you want to define can follow the `VARIABLE` keyword. For example:

```
VARIABLE
   string activity_name;
   float a := 3.243;
   activity act_id;
```

Value assignments are optional. If they are assigned, they represent the default value of the variable at the first generation of a particular document. The value can be any expression that does not contain other variables or parameters.

Variables that are declared as Statemate elements and list of items cannot be assigned initial values.

Syntax

```
VARIABLE
type identifier [:= value] [, identifier [:= value],...;
[type identifier [:= value],...;]
.
.
.
```

Parameters

Parameter	Description
`type`	The parameter type
`identifier`	The name of the variable
`variable`	The initial value of the variable

Notes

The assignment of initial values is optional. An initial value is an expression that cannot contain other variables. Initial values are allowed only for integer, float, and string.

Example

```
VARIABLE
    INTEGER         status := 0;
    STATE           statechart;
    LIST OF STATE   sub_list;
    STATE           sub_state, parent;
    STRING          name;
```

See Also

- **ASSIGNMENT**
- **CONSTANT**
- **PARAMETER**
- **TEMPLATE Statement**

VERBATIM

Description

Passes text literally to an output file.

When the verbatim symbols `/@` and `@/` frame text in the template file, the text is passed literally (without interpretation) to the output segment file. Comments inside the frame, rather than being ignored, are also passed literally. The end-of-statement character, ";", is not required following the concluding verbatim symbol.

Verbatim statements can be used to pass the following to the output file:

- Formatting commands applicable to a specific formatter.
- Short text passages such as titles, opening remarks, and so on. Despite the absence of any length restriction on verbatim text, longer text passages are usually passed using the `INCLUDE` file statement.

Syntax

```
/@ verbatim_text @/
```

Parameters

Parameter	Description
`verbatim_text`	One or more lines of printable text characters

Example

```
Consider the following verbatim section:

          SEGMENT section1;
          BEGIN      /@
          .title ACTIVITY-SPEC
          .skip 2
          .center; AN ACTIVITY SPECIFICATION
          -- This section will describe the
          -- purpose of the activities.
          @/
          -- this line will not appear in the output
          END;
```

When the template is executed, the resulting output is as follows:

```
          title ACTIVITY-SPEC
          .skip 2
          .center; AN ACTIVITY SPECIFICATION
          -- This section will describe the
          -- purpose of the activities.
```

See Also

- **EXECUTE**
- **INCLUDE**
- **REPORT**
- **TABLE**
- **WRITE**
- **READ Statement**

WHILE/LOOP

Description

Provides iterative execution of DGL statements. The execution of statements is determined by evaluation of the `boolean_expression`.

The statements are executed until the expression evaluates to false. For example:

```
WHILE  a > b  LOOP
       .
       .
   b := b + k;
       .
       .
END LOOP;
```

The statements between the keywords `LOOP` and `END LOOP` are executed as long as `a` is greater than `b`.

Assume that `b` changes its value inside the loop and in one of the iterations the expression `a > b` becomes false. In the next iteration, the expression is examined and, because `a > b` is now `FALSE`, the execution of template statements continues with the first statement after the `END LOOP`.

Syntax

```
WHILE boolean_expression LOOP
   statements
END LOOP;
```

Parameters

Parameter	Description
`boolean_expression`	The Boolean expression used to test the conditions
`statements`	One or more DGL statements

Notes

The value of `boolean_expression` must be altered within the loop, or an `EXIT` statement must be executed in order to terminate the loop.

Example

```
WHILE count < 25 LOOP
    num_control := alpha / 5;
    WRITE (count, num_control, '\n';
    alpha := synch +7;
    count := count + 1;
END LOOP;
```

See Also

- **EXIT**
- **FOR/LOOP**
- **IF/THEN/ELSE**
- **SELECT/WHEN**
- **STOP**
- **Control Flow Statements**

WRITE

Description

Writes expression values to any of the following:

- The document output segment
- Another file
- The dialog area of the tool window

You can write a numeric or string expression that is evaluated in the template, or a literal piece of text. In addition, the `WRITE` statement can be used to write information retrieved from the database, such as element names.

The `WRITE` statement is commonly used to write lines that include text (string literals) together with expression values. For example:

```
WRITE ('NAME:', di_name);
```

- This results in KUKU being written in the output segment file NAME, where KUKU is a value of `di_name`.

  Note the following:

  – There can be more than one write expression. When there are multiple expressions, they are separated by commas.

  – For lines of pure text, it is better to use the `Verbatim` statement.

  – Literal strings can include the formatting characters

`\n`	New line
`\t`	Tab

- For example, the following call writes the value of `alpha` at the beginning of the next line in the output segment file:

  ```
  WRITE ('\n', alpha);
  ```

- Optionally, you can specify the minimum number of characters to be written in the output file using the following syntax:

  ```
  expression : num
  ```

- In this syntax, `expression` can be either a numeric or a string expression, and `num` is an integer constant or integer expression that represents the minimum number of characters that `expression` will occupy. `expression` and `num` can involve operands, operations, and function calls.

- For example:

```
WRITE (act_name: 10, ',' , act_synonym);
```

- This call results in the string value for `act_name` being written in the output file to a length of at least 10 characters; if the name has less than this number, blanks are added to achieve the specified string length. For example:

```
COMP   , SET
```

- In this example, spaces have been added to "COMP" to give it a length of 10 characters.
- The use of `num` determines the minimum number of characters to be written in the output file, as follows:
- For a string, the length of the string is the minimum number of output characters. When specified, and where `num` is greater than the string length, blanks are padded to the right of the string to achieve a total string length of `num`.
- For an integer, when `num` is specified, and where `num` is greater than the number of digits in the integer, blanks are padded to the left of the number to achieve a total output length of `num`.
- For a real number, when `num` is *not* specified, the value is output to no more than 8 decimal places. Thereafter, the number is automatically rounded. When specified, and where `num` is greater than the digits output according to the default, blanks are padded to the left of the number to achieve a total output length of `num`. Where `num` is less than the digits output according to the default, the decimal portion of the number might be rounded to arrive at a specified output length of `num`. However, in no case will the integer portion of the real number be truncated.

Syntax

```
WRITE ([fl,] write_expression,...);
```

Parameters

Parameter	Description
`f1`	An identifier of type `FILE`, previously assigned by the `OPEN` statement, to which to write the specified string. Using the `WRITE` statement to write to a file or to the dialog area is particularly useful if you want to write messages (error messages, run-time messages, and so on). When writing to a file or to the dialog area, you must include the `fl` identifier. In such cases, you must also precede the `WRITE` statement with an `OPEN` statement.
`write_expression`	The expression value to write to the document segment, file, or dialog area.

Example

```
WRITE ('Name:':8, name,'\n','Value:':8, v);
```

Assume that `name` contains "Xfactor" and `v` is an integer that equals 5105. This call writes the following lines to the output file:

```
Name:   Xfactor
Value:  5105
```

Note: The `WRITE` command cannot access a `list` variable directly; if you attempt to write a variable that refers to a list, the Documentor displays an error message. To output a list, use a control flow construct such as a loop, writing one list item at a time.

Using WRITE to Produce Messages

You can use a `WRITE` statement to write information to a file or to the dialog area, instead of to the document itself. For this, you must first open a file in `OUTPUT` mode using the `OPEN` statement (refere to **OPEN**).

To write a string message to a file, use the following syntax:

```
WRITE (fl, write_expression);
```

In the syntax, `fl` is a pointer to the file to which you want to write the messages.

See Also

- **EXECUTE**
- **INCLUDE**
- **REPORT**
- **TABLE**
- **VERBATIM**
- **READ Statement**

Query Functions

This section documents each query function, its purpose, and other notes. The functions are organized into sections by “element types returned by functions.” Within each section, functions are organized by “type of element in the input list.” Each function description includes the query from the Query Model tool of Statemate that matches it, if applicable.

The topics are as follows:

- **Calling Query Functions**
- **Query Function Input Arguments**
- **Examples of Query Functions**

Query functions extract lists of elements from the database that conform to a specific criterion.

The search enables you to query the Statemate database. This tool uses a comprehensive set of predefined queries to obtain information. All these queries operate on a list of Statemate elements, called the *input list*. Each query generates an output list of elements that meet a criterion designated by the specific query. Generally, elements in the output list are related to elements in the input list in one of two ways:

- The output list is a subset of input list elements that have a specific characteristic. For example, the output list consists of all And-states in the input list.
- Elements in the output list fulfill a specific relationship to elements in the input list. For example, the output list consists of all states that are descendants of states in the input list.

Most query functions correspond to queries from the search. These functions give you the same information that the corresponding queries do. Most functions require you to provide an input list as an input argument. This input list generally consists of elements of a particular type. The function returns a list of elements of the same or different type (as the input list).

The retrieval process is as follows:

1. Generate the input list.
2. Specify the query and input list. Receive the input list. Note that other procedures may be performed before you use the retrieved information.
3. Use the output list.

Calling Query Functions

Most of the query functions use the following calling sequence:

```
stm_r_yy_relation_xx (xx_list, status)
```

In this syntax:

- **stm_r_**—Designates the function as a Statemate database retrieval function.
- **yy**—The two-character type abbreviation for elements in the output list.
- **relation**—The relationship between the input and output lists (describes the query to be applied to the input list).
- **xx**—The two-character type abbreviation for elements in the input list.
- **xx_list**—The input list to the function.
- **status**—The return function status code. There are three possible status codes: `stm_success`, `stm_nil_list`, and `stm_missing_element_in_list`.

For example:

```
stm_r_st_and_st (state_list, status)
```

This function returns the states from the input list `state_list` that are and-states.

The following function returns the activities performed throughout the states in `state_list`:

```
stm_r_st_ac_throughout_st (state_list, status)
```

The following sections document the query functions that use a different calling sequence.

By Attributes

The `by_attributes` function returns all elements in the input list that have an attribute `attr_name`, whose value is `attr_val`.

The syntax is as follows:

```
stm_r_xx_by_attributes_xx (xx_list, attr_name, attr_val, status)
```

In this syntax:

- **stm_r_**—Designates the function as a Statemate database retrieval function.
- **xx**—The two-character type abbreviation for elements in the input and output lists.
- **by_attributes**—The criterion to be met by elements in the input list.
- **xx_list**—The input list to the function.
- **attr_name**—A pattern for the attribute name.

- **attr_val**—A pattern for the attribute value to be matched.
- **status**—The return function status code. There are three possible status codes: `stm_success`, `stm_nil_list`, and `stm_missing_element_in_list`.

For example:

```
stm_r_md_by_attributes_md (module_list, 'LANGUAGE',
    'PASCAL', status)
```

This function returns all modules in `module_list` that have an attribute `LANGUAGE`, whose value is `PASCAL`.

By Structure Type

The `by_structure_type` function returns all elements in the input list that have a structure type `xx_structure_type`.

The syntax of the `by_structure_type` function is as follows:

```
stm_r_xx_by_structure_type_xx (xx_list, xx_structure_type, status)
```

In this syntax:

- **stm_r_**—Designates the function as a Statemate database retrieval function
- **xx**—The two-character type abbreviation for elements in the input and output list
- **by_structure_type**—The structure type referenced
- **xx_list**—The input list to the function
- **xx_structure_type**—The structure type referenced (array, single, or queue in the element's form)
- **status**—The return function status code

For example:

```
stm_r_di_by_structure_type_di (di_list, stm_di_array, status)
```

This function returns all data items in `di_list` that have an array structure type.

Name and Synonym Patterns

The `name_of` and `synonym_of` functions search the entire database for elements whose name (or synonym) matches the pattern specified in the argument pattern.

The syntax is as follows:

```
stm_r_xx_name_of_xx (pattern, status)
stm_r_xx_synonym_of_xx (pattern, status)
```

In this syntax:

- **stm_r_**—Designates the function as a Statemate database retrieval function
- **xx**—The two-character type abbreviation of elements in the output list
- **name_of** or **synonym_of**—The criterion to be met (specifies that the query “Element whose name matches a pattern” or “Element whose synonym matches a pattern” is to be applied)
- **pattern**—A character string you supply as an input argument
- **status**—The return function status code

For example:

```
stm_r_ev_name_of_ev ('EV*', status)
```

This function returns all events from the database whose names begin with the string `EV`.

Query Function Input Arguments

Most query functions operate on an input list that you specify. The following table lists the arguments for query functions.

Argument	Description	DGL Data Type
*xx*_list	The input list of elements upon which to perform the query.	List of Statemate element type (for example, `LIST OF STATE`, `LIST OF ELEMENT`, and so on).
attribute name	The pattern for the name of an attribute defined in the attribute field of a Statemate element form. Two special characters can be used as wildcards; the question mark (?) and the asterisk (*). A question mark indicates that any single character can occupy that position. An asterisk indicates that any number of characters (including 0) can occupy that position.	String
attribute value	The pattern for the value of an attribute defined in the attribute field of a Statemate element form. The pattern can include wildcards (? and *).	String
pattern	An alphanumeric string to match a Statemate element name (or synonym). The pattern can include wildcards (? and *).	String

Examples of Query Functions

This section shows how to use query function calls to perform common tasks.

Query Function Example 1

The following example shows how to build an input list for the query functions:

```
VARIABLE
   ACTIVITY          act_id;
   LIST OF ACTIVITY  act_list;
   INTEGER           status;
                     .
                     .
act_id := stm_r_ac ('A1', status);
act_list := stm_r_ac_physical_sub_of_ac ({act_id},
   status);
                     .
                     .
```

The variable act_id contains the ID of the activity A1. You built the input list by enclosing act_id in curly braces ({}). In this case, the input list consists of one element. If you want to build an input list of multiple elements, enclose them all in curly braces, separated by commas.

Note

The input list is built from element IDs, *not* element names.

Query Function Example 2

The following example shows how you use query functions in succession. Assume that you want to know all the basic states that are descendants of the state `S1`.

```
VARIABLE
   STATE           st_id;
   INTEGER         status;
   LIST OF STATE   descen_states, basic_states;
                   .
                   .
st_id := stm_r_st ('S1', status);
descen_states := stm_r_st_physical_desc_of_st (
   {st_id}, status);
basic_states := stm_r_st_basic_st (descen_states,
   status);
                   .
                   .
```

Note that `descen_states` is *not* enclosed in braces because its value comprises a list of states.

Query Function Example 3

The following example shows how to use query function calls in a place where lists can appear (for example, in FOR statements):

```
VARIABLES
   EVENT       e;
   INTEGER     status;
               .
               .
FOR e IN stm_r_ev_name_of_ev ('EV*', status)
   LOOP
      WRITE ('\n', stm_r_ev_name (e, status));
END LOOP;
```

This example writes out the name of all the events in the database whose name begins with the string EV.

List of Query Functions

The query functions are grouped alphabetically first by output list type, then by input list type. The output types are as follows:

- **Activities (ac)**
- **A-Flow-Lines (af, ba, laf)**
- **Actions (an)**
- **Charts (ch)**
- **Connectors (cn)**
- **Conditions (co)**
- **Data-Items (di)**
- **Data-Stores (ds)**
- **User-Defined Types (dt)**
- **Events (ev)**
- **Fields (fd)**
- **Functions (fn)**
- **Information-Flows (if)**
- **M-Flow-Lines (bf, bm, lmf, mf)**
- **Modules (md)**
- **Mixed (mx)**
- **Routers (router)**

- Subroutines (sb)
- States (st)
- Timing Constraint (tc)
- Transitions (tr)

Activities (ac)

This section documents the query functions that return a list of activities.

Input List Type: ac

stm_r_ac_associates_uc	**Query:** Activities by use cases **Purpose**: This API returns the activities that associate with use-cases in the input list **Syntax:** `stm_r_ac_associates_uc(IN uc_lst: LIST OF USE_CASE, OUT status: INTEGER)`
stm_r_ac_basic_ac	**Query:** Basic activities **Purpose:** Returns the activities in the input list that have no descendants **Syntax:** `stm_r_ac_basic_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_ac_by_attributes_ac	**Query:** Activities by attributes **Purpose:** Returns the activities in the input list that match the specified attribute name and value **Syntax:** `stm_r_ac_by_attributes_ac (IN ac_list: LIST OF ACTIVITY, IN attr_name: STRING, IN attr_value: STRING, OUT status: INTEGER)`
stm_r_ac_callback_binding_ac	**Query:** Activities with callback bindings **Purpose:** Returns the activities in the input list that have callback bindings **Syntax:** `stm_r_ac_callback_binding_ac (IN el_list: LIST OF ACTIVITY, OUT status: INTEGER)`

stm_r_ac_component_instance_ac	**Query:** Activities that are instances of components **Purpose:** Returns the activities in the input list that have instances of components **Syntax:** `stm_r_ac_component_instance_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_ac_continuous_instance_ac	**Query:** Activities with continuous instances **Purpose:** Returns the activities in the input list that have continuous instances **Syntax:** `stm_r_ac_continuous_instance_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_ac_control_ac	**Query:** Control activities **Purpose:** Returns the activities in the input list that are control activities **Syntax:** `stm_r_ac_control_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_ac_control_terminated_ac	**Query:** Controlled-terminated activities **Purpose:** Returns the activities in the input list that are control-terminated activities **Syntax:** `stm_r_ac_control_terminated_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_ac_data_store_ac	**Query:** Data-stores **Purpose:** Returns the activities in the input list that are data-stores **Syntax:** `stm_r_ac_data_store_ac (IN el_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_ac_def_of_instance_ac	**Query:** Definition activities of a given activity **Purpose:** Returns the definition activities (top-level in the definition chart) for instances in the input list **Syntax:** `stm_r_ac_def_of_instance_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_ac_defined_environment_ac	**Query:** Environment activities **Purpose:** Returns the activities in the input list that were defined as environment activities **Syntax:** `stm_r_ac_defined_environment_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`

stm_r_ac_explicit_defined_ac	**Query:** Activities explicitly defined **Purpose:** Returns from the input list those activities that were explicitly defined **Syntax:** `stm_r_ac_explicit_defined_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_ac_external_ac	**Query:** External activities **Purpose:** Returns the activities in the input list that are external **Syntax:** `stm_r_ac_external_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_ac_generic_instance_ac	**Query:** Generic instance activities **Purpose:** Returns the activities in the input list that are instances of generic charts **Syntax:** `stm_r_ac_generic_instance_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_ac_imp_best_match_ac	**Query:** Activities whose selected implementation is **Best Match** **Purpose:** Returns the activities in the input list that are implemented as the **Best Match** using **Select Implementation** in the properties **Syntax:** `stm_r_ac_imp_best_match_ac (IN el_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_ac_imp_mini_spec_ac	**Query:** Activities implemented in a mini-spec **Purpose:** Returns the activities in the input list that are implemented in a mini-spec **Syntax:** `stm_r_ac_imp_mini_spec_ac (IN el_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_ac_imp_none_ac	**Query:** Activities whose selected implementation is **None** **Purpose:** Returns the activities in the input list that are not implemented **Syntax:** `stm_r_ac_imp_none_ac (IN el_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_ac_imp_sb_bind_ac	**Query:** Activities implemented with subroutine bindings **Purpose:** Returns the activities in the input list that are implemented as **Subroutine Binding** using **Select Implementation** in the properties **Syntax:** `stm_r_ac_imp_sb_bind_ac (IN el_list: LIST OF ACTIVITY, OUT status: INTEGER)`

stm_r_ac_imp_truth_table_ac	**Query:** Activities implemented in a truth table **Purpose:** Returns the activities in the input list that were implemented in a truth table **Syntax:** `stm_r_ac_imp_truth_table_ac (IN el_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_ac_instance_ac	**Query:** Instance activities **Purpose:** Returns those activities in the input list that are instances **Syntax:** `stm_r_ac_instance_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_ac_instance_of_def_ac	**Query:** Instance activities of a given definition activity **Purpose:** Returns the instance activities for definition activities (top-level activities in a definition chart) in the input list **Syntax:** `stm_r_ac_instance_of_def_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_ac_internal_ac	**Query:** Internal activities **Purpose:** Returns the activities in the input list that are internal activities (not external or control) **Syntax:** `stm_r_ac_internal_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_ac_is_occurrence_of_ac	**Query:** Activity occurrences of a given activity **Purpose:** Returns the activities for which the activities in the input list appear in the **Is activity** field of their form **Syntax:** s**tm_r_ac_is_occurrence_of_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER**)
stm_r_ac_is_principal_of_ac	**Query:** Principal activities of a given activity **Purpose:** Returns the activities appearing in the **Is activity** field of the activities in the input list **Syntax:** `stm_r_ac_is_principal_of_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_ac_logical_desc_of_ac	**Query:** Logical descendants of a given activity **Purpose:** Returns the logical descendants of the activities in the input list, taking into account the translation of instances to their definition charts **Syntax:** `stm_r_ac_logical_desc_of_ ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`

stm_r_ac_logical_parent_of_ac	**Query:** Logical parent activities of a given activity **Purpose:** Returns the logical parent activities of the activities in the input list, taking into account the translation of instances to their definition charts **Syntax:** `stm_r_ac_logical_parent_of_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_ac_logical_sub_of_ac	**Query:** Logical subactivities of a given activity **Purpose:** Returns the logical subactivities of the activities in the input list, taking into account the translation of instances to their definition charts **Syntax:** `stm_r_ac_logical_sub_of_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_ac_mini_spec_ac	**Query:** Activities having mini-specs **Purpose:** Returns the activities in the input list that have a mini-spec **Syntax:** `stm_r_ac_mini_spec_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_ac_name_of_ac	**Query:** Activities whose names match a given pattern **Purpose:** Returns all the activities whose names match a given pattern **Syntax:** `stm_r_ac_name_of_ac (IN pattern: STRING, OUT status: INTEGER)`
stm_r_ac_offpage_instance_ac	**Query:** Offpage instance activities **Purpose:** Returns the activities in the input list that are instances of offpage charts **Syntax:** `stm_r_ac_offpage_instance_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_ac_physical_desc_of_ac	**Query:** Physical descendants of a given activity **Purpose:** Returns the physical descendants (those within the same chart) for the activities in the input list **Syntax:** `stm_r_ac_physical_desc_of_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_ac_physical_parent_of_ac	**Query:** Physical parent activities of a given activity **Purpose:** Returns the physical parent activities (those within the same chart) for the activities in the input list **Syntax:** `stm_r_ac_physical_parent_of_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`

stm_r_ac_physical_sub_of_ac	**Query:** Physical subactivities of a given activity **Purpose:** Returns the physical subactivities (those within the same chart) for the activities in the input list **Syntax:** `stm_r_ac_physical_sub_of_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_ac_procedure_like_ac	**Query:** Procedure-like activities **Purpose:** Returns the activities in the input list that are procedure-like activities **Syntax:** `stm_r_ac_procedure_like_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_ac_resolved_to_ext_ac	**Query:** Activities resolved to a given external activity **Purpose:** Returns the activities (internal, external, or environment) to which the external activities in the input list are resolved **Syntax:** `stm_r_ac_resolved_to_ext_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_ac_self_terminated_ac	**Query:** Self-terminated activities **Purpose:** Returns the activities in the input list that are self-terminated **Syntax:** `stm_r_ac_self_terminated_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_ac_subroutine_binding_ac	**Query:** Activities with subroutine bindings **Purpose:** Returns the activities in the input list that have subroutine bindings (regardless of the implementation setting in the properties) **Syntax:** `stm_r_ac_subroutine_binding_ac (IN el_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_ac_synonym_of_ac	**Query:** Activities whose synonyms match a given pattern **Purpose:** Returns all the activities whose synonyms match the specified pattern **Syntax:** `stm_r_ac_synonym_of_ac (IN pattern: STRING, OUT status: integer)`
stm_r_ac_unresolved_ac	**Query:** Unresolved activities **Purpose:** Returns the unresolved activities in the input list **Syntax:** `stm_r_ac_unresolved_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`

stm_r_ac_ext_ll_ac	**Query**: External life-line activities **Purpose:** Returns the External life-line activities in the input list **Syntax**: `stm_r_ac_ext_ll_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_ac_lifeline_ac	**Query**: Life-line activities **Purpose**: Returns the life-line activities in the input list **Syntax**: `stm_r_ac_lifeline_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_ac_boundary_box_ac	**Query**: Boundary-box actvities **Purpose**: Returns the Boundary-box actvitites in the input list **Syntax**: `stm_r_ac_boundary_box_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_ac_use_case_ac	**Query**: Use-case activities **Purpose**: Returns the Use-case activities in the input list **Syntax**: `stm_r_ac_use_case_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_ac_actor_ac	**Query**: `Actor activities` **Purpose**: Returns the Actor activities in the input list **Syntax**: `stm_r_ac_actor_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_ac_router_ac	**Query**: Router sctivities **Purpose**: Returns the Router activiites in the input list **Syntax**: `stm_r_ac_router_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_ac_external_router_ac	**Query**: External router activities **Purpose**: Returns the Router activiites in the input list **Syntax**: `stm_r_ac_external_router_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`

Input List Type: af

stm_r_ba_defined_in_ch	**Query:** Activities that are sources for a given a-flow-line
	Purpose: Returns the A-Flow-lines defined in the input list of charts.
	Syntax:
	`STM_R_BA_DEFINED_IN_CH(IN ch_list: LIST OF CHART, OUT`
	`status: INTEGER):LIST OF A_FLOW_LINE;`

stm_r_ba_enter_ds	**Query:** Activities that are sources for a given a-flow-line **Purpose:** Returns the A-Flow-lines entering the Data-Stores in the input list. **Syntax:** `STM_R_BA_ENTER_DS(IN in_list: LIST OF DATA_STORE, OUT status: INTEGER):LIST OF A_FLOW_LINE;`
stm_r_ba_exit_from_ds	**Query:** Activities that are sources for a given a-flow-line **Purpose:** Returns the activities that are sources of a-flow-lines in the input list **Syntax:** `STM_R_BA_EXIT_FROM_DS (IN in_list: LIST OF DATA_STORE, OUT status: INTEGER):LIST OF A_FLOW_LINE;`
stm_r_ba_enter_an	**Query:** Activities that are sources for a given a-flow-line **Purpose:** Returns the activities that are sources of a-flow-lines in the input list **Syntax:** `STM_R_BA_ENTER_AN(IN in_list: LIST OF ACTIVITY, OUT status: INTEGER):LIST OF A_FLOW_LINE;`
stm_r_ba_exit_from_ac	**Query:** Activities that are sources for a given a-flow-line **Purpose:** Returns the activities that are sources of a-flow-lines in the input list **Syntax:** `STM_R_BA_EXIT_FROM_AC(IN in_list: LIST OF ACTIVITY, OUT status: INTEGER):LIST OF A_FLOW_LINE;`
stm_r_ba_enter_cn	**Query:** Activities that are sources for a given a-flow-line **Purpose:** Returns the activities that are sources of a-flow-lines in the input list **Syntax:** `STM_R_BA_ENTER_CN(IN in_list: LIST OF CONNECTOR, OUT status: INTEGER):LIST OF A_FLOW_LINE;`

stm_r_ba_exit_from_cn	**Query:** Activities that are sources for a given a-flow-line **Purpose:** Returns the activities that are sources of a-flow-lines in the input list **Syntax:** `STM_R_BA_EXIT_FROM_CN(IN in_list: LIST OF CONNECTOR, OUT status: INTEGER):LIST OF A_FLOW_LINE;`
stm_r_laf_containing_ba	**Query:** Activities that are sources for a given a-flow-line **Purpose:** Returns the activities that are sources of a-flow-lines in the input list **Syntax:** `STM_R_LAF_CONTAINING_BA(IN ba_list: LIST OF A_FLOW_LINE, OUT status: INTEGER):LIST OF A_FLOW_LINE`
stm_r_ba_contained_in_af	**Query:** Activities that are sources for a given a-flow-line **Purpose:** Returns the activities that are sources of a-flow-lines in the input list **Syntax:** `STM_R_BA_CONTAINED_IN_AF(IN af_list: LIST OF A_FLOW_LINE, OUT status: INTEGER):LIST OF A_FLOW_LINES`
stm_r_ac_source_of_af	**Query:** Activities that are sources for a given a-flow-line **Purpose:** Returns the activities that are sources of a-flow-lines in the input list **Syntax:** `stm_r_ac_source_of_af (IN af_list: LIST OF A_FLOW_LINE, OUT status: INTEGER)`
stm_r_ac_target_of_af	**Query:** Activities that are targets of a given a-flow-line **Purpose:** Returns the activities that are targets for a-flow-lines in the input list **Syntax:** `stm_r_ac_target_of_af (IN af_list: LIST OF A_FLOW_LINE, OUT status: INTEGER)`

Input List Type: ch

stm_r_ac_def_or_unres_in_ch	**Query:** Activities defined or unresolved in a given chart **Purpose:** Returns activities that are explicitly defined or unresolved in the charts of the input list **Syntax:** `stm_r_ac_def_or_unres_in_ ch (IN ch_list: LIST OF CHART,OUT status: INTEGER)`
stm_r_ac_defined_in_ch	**Query:** Activities defined in a given chart **Purpose:** Returns the activities that are explicitly defined in the charts of the input list **Syntax:** `stm_r_ac_defined_in_ch (IN ch_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_ac_described_by_ch	**Query:** Control activities described by a given statechart **Purpose:** Returns the control activities described by statecharts in the input list **Syntax:** `stm_r_ac_described_by_ch (IN ch_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_ac_instance_of_ch	**Query:** Activities instance of a given chart **Purpose:** Returns the instance activities defined by the charts in the input list **Syntax:** `stm_r_ac_instance_of_ch (IN ch_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_ac_root_in_ch	**Query:** Root activities of a given chart **Purpose:** Returns the internally defined activities (of type diagram) attached to the charts in the input list **Syntax:** `stm_r_ac_root_in_ch (IN ch_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_ac_top_level_in_ch	**Query:** Top-level activities of a given chart **Purpose:** Returns the top-level activities (not contained in any box) of the charts in the input list **Syntax:** `stm_r_ac_top_level_in_ch (IN ch_list: LIST OF CHART,OUT status: INTEGER)`

stm_r_ac_unresolved_in_ch	**Query:** Activities unresolved in a given chart **Purpose:** Returns activities that are unresolved in the charts of the input list **Syntax:** `stm_r_ac_unresolved_in_ch (IN ch_list: LIST OF CHART,OUT status: INTEGER)`

Input List Type: ds

stm_r_ac_parent_of_ds	**Query:** Parent activities of a given data-store **Purpose:** Returns the activities that encapsulate the specified data-stores from the input list **Syntax:** `stm_r_ac_parent_of_ds (IN ds_list: LIST OF DATA_STORE, OUT status: INTEGER)`

Input List Type: md

stm_r_ac_carried_out_by_md	**Query:** Activities carried out by a given module. **Purpose:** Returns the activities carried out by modules in the input list. The module appears in the **Implemented by Module** field of the activity's form. **Syntax:** `stm_r_ac_carried_out_by_md (IN md_list: LIST OF MODULE, OUT status: INTEGER)`

Input List Type: mx

stm_r_ac_affecting_mx	**Query:** Activities in which a given element is affected. **Purpose:** Returns the activities that affect (modify, generate, or activate) the elements (for example, events, data-items, or activities) in the input list. **Syntax:** stm_r_ac_affecting_mx (IN mx_list: LIST OF ELEMENT,OUT status: INTEGER)
stm_r_ac_meaningly_affecting_mx	**Query:** Activities in which a given element is affected. **Purpose:** Identical to stm_r_ac_affecting_mx, but when the input list includes an ID of a record/union, stm_r_ac_meaningly_affecting_mx will also return elements that affect a field of the record/union, and not necessarily the whole record/union element. **Syntax**: `stm_r_ac_meaningly_affecting_mx (stm_list in_list, int *status);`
stm_r_ac_using_mx	**Query:** Activities in which a given element is used. **Purpose:** Returns the activities that use (evaluate) the elements (basic events, conditions, data-items, states, and activities) in the input list. **Syntax:** `stm_r_ac_using_mx (IN mx_list: LIST OF ELEMENT,OUT status: INTEGER)`
stm_r_ac_meaningly_using_mx	**Query:** Activities in which a given element is used. **Purpose:** Identical to stm_r_ac_using_mx, but when the input list includes an ID of a record/union, stm_r_ac_meaningly_using_mx will also return elements that use a field of the record/union, and not necessarily the whole record/union element **Syntax:** `stm_r_ac_meaningly_using_mx (stm_list in_list, int *status);`

Input List Type: router

stm_r_ac_parent_of_router	**Query:** Parent activities of a given router **Purpose:** Returns the activities that encapsulate the specified routers from the input list **Syntax:** `stm_r_ac_parent_of_router (IN router_list: LIST OF ROUTER, OUT status: INTEGER)`

Input List Type: st

stm_r_ac_throughout_st	**Query:** Activities performed throughout a given state **Purpose:** Returns the activities performed throughout states in the input list (as defined in the **Activities Within/Throughout** field of the state's form) **Syntax:** `stm_r_ac_throughout_st (IN st_list: LIST OF STATE, OUT status: INTEGER)`
stm_r_ac_within_st	**Query:** Activities performed within a given state **Purpose:** Returns the activities performed within states in the input list (as defined in the **Activities Within/Throughout** field of the state's form) **Syntax:** `stm_r_ac_within_st (IN st_list: LIST OF STATE, OUT status: INTEGER)`

A-Flow-Lines (af, ba, laf)

This section lists the query functions that return a list of a-flow-lines.

Two abbreviations are used in these functions:

- `af`—Global (compound) a-flow-lines
- `ba`—Basic a-flow-lines
- `laf`—Local a-flow-lines

Output List Type: af

Input List Type: ac

stm_r_af_from_source_ac	**Query:** A-flow-lines whose source is a given activity **Purpose:** Returns global compound a-flow-lines that originate at activities in the input list **Syntax:** `stm_r_af_from_source_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_af_input_to_ac	**Query:** A-flow-lines input to a given activity within chart **Purpose:** Returns all local compound a-flow-lines that originate outside and terminate at (or inside) activities in the input list **Syntax:** `stm_r_af_input_to_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_af_output_from_ac	**Query:** A-flow-lines output from a given activity **Purpose:** Returns all global compound a-flow-lines that originate at (or inside) and terminate outside activities in the input list **Syntax:** `stm_r_af_output_from_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_af_to_target_ac	**Query:** A-flow-lines whose target is a given activity **Purpose:** Returns global a-flow-lines that terminate at activities in the input list **Syntax:** `stm_r_af_to_target_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`

Input List Type: ba

stm_r_af_containing_ba	**Query:** A-flow lines from an input list. **Purpose:** Returns the a-flow-lines that contain the basic a-flow-lines in the input list. **Syntax:** `STM_R_AF_CONTAINING_BA(IN ba_list: LIST OF A_FLOW_LINE, OUT status:INTEGER):LIST OF A_FLOW_LINE`

Input List Type: co

stm_r_af_within_flows_co	**Query:** A-flow-lines through which a given condition flows **Purpose:** Returns the a-flow-lines through which conditions in the input list actually flow **Syntax:** `stm_r_af_within_flows_co (IN co_list: LIST OF CONDITION, OUT status: INTEGER)`
stm_r_af_within_labels_co	**Query:** A-flow-lines labeled with a given condition **Purpose:** Returns the a-flow-lines labeled with conditions in the input list **Syntax:** `stm_r_af_within_labels_co (IN co_list: LIST OF CONDITION, OUT status: INTEGER)`

Input List Type: di

stm_r_af_within_flows_di	**Query:** A-flow-lines through which a given data-item flows **Purpose:** Returns the a-flow-lines through which data-items in the input list actually flow **Syntax:** `stm_r_af_within_flows_di (IN di_list: LIST OF DATA_ITEM, OUT status: INTEGER)`
stm_r_af_within_labels_di	**Query:** A-flow-lines labeled by a given data-item **Purpose:** Returns the a-flow-lines labeled with data-items in the input list **Syntax:** `stm_r_af_within_labels_di (IN di_list: LIST OF DATA_ITEM, OUT status: INTEGER)`

Input List Type: ds

stm_r_af_from_source_ds	**Query:** A-flow-lines whose source is a given data-store **Purpose:** Returns global compound a-flow-lines that originate at data-stores in the input list **Syntax:** `stm_r_af_from_source_ds (IN ds_list: LIST OF DATA_STORE, OUT status: INTEGER)`
stm_r_af_to_target_ds	**Query:** A-flow-lines whose target is a given data-store **Purpose:** Returns global compound a-flow-lines that terminate at data-stores in the input list **Syntax:** `stm_r_af_to_target_ds (IN ds_list: LIST OF DATA_STORE, OUT status: INTEGER)`

Input List Type: ev

stm_r_af_within_flows_ev	**Query:** A-flow-lines through which a given event flows **Purpose:** Returns the a-flow-lines through which events in the input list actually flow **Syntax:** `stm_r_af_within_flows_ev (IN ev_list: LIST OF EVENT, OUT status: INTEGER)`
stm_r_af_within_labels_ev	**Query:** A-flow-lines through which a given event flows **Purpose:** Returns the a-flow-lines labeled with events in the input list **Syntax:** `stm_r_af_within_labels_ev (IN ev_list: LIST OF EVENT, OUT status: INTEGER)`

Input List Type: if

stm_r_af_within_flows_if	**Query:** A-flow-lines through which a given information-flow flows **Purpose:** Returns the a-flow-lines through which information-flows in the input list actually flow **Syntax:** `stm_r_af_within_flows_if (IN if_list: LIST OF INFORMATION_FLOW, OUT status: INTEGER)`
stm_r_af_within_labels_if	**Query:** A-flow-lines labeled with a given information-flow **Purpose:** Returns the a-flow-lines labeled with information-flows in the input list **Syntax:** `stm_r_af_within_labels_if (IN if_list: LIST OF INFORMATION_FLOW, OUT status: INTEGER)`

Input List Type: laf

stm_r_af_containing_laf	**Query:** None **Purpose:** Returns the global a-flow-lines (which might spread over several charts) that contain the local a-flow-lines (those within charts) in the input list **Syntax:** `stm_r_af_containing_laf (IN af_list: LIST OF A_FLOW_LINE, OUT status: INTEGER)`

Input List Type: mx

stm_r_af_from_source_mx	**Query:** A-flow-lines whose source is a given element **Purpose:** Returns global compound a-flow-lines whose source is an element from the input list **Syntax:** `stm_r_af_from_source_mx (IN mx_list: LIST OF ELEMENT,OUT status: INTEGER)`
stm_r_af_to_target_mx	**Query:** A-flow-lines whose target is given element **Purpose:** Returns global compound a-flow-lines whose target is an element from the input list **Syntax:** `stm_r_af_to_target_mx (IN mx_list: LIST OF ELEMENT,OUT status: INTEGER)`
stm_r_af_within_flows_mx	**Query:** A-flow-lines through which a given element flows **Purpose:** Returns the a-flow-lines through which elements in the input list actually flow **Syntax:** `stm_r_af_within_flows_mx (IN mx_list: LIST OF ELEMENT,OUT status: INTEGER)`
stm_r_af_within_labels_mx	**Query:** A-flow-lines labeled by a given element **Purpose:** Returns the a-flow-lines labeled with elements in the input list **Syntax:** `stm_r_af_within_labels_mx (IN mx_list: LIST OF ELEMENT,OUT status: INTEGER)`

Input List Type: router

stm_r_af_from_source_router	**Query:** A-flow-lines whose source is a given router **Purpose:** Returns global compound a-flow-lines whose source is a router from the input list **Syntax:** `stm_r_af_from_source_router (IN router_list: LIST OF ROUTER, OUT status: INTEGER)`
stm_r_af_to_target_router	**Query:** A-flow-lines whose target is given router **Purpose:** Returns global compound a-flow-lines whose target is a router from the input list **Syntax:** `stm_r_af_to_target_router (IN router_list: LIST OF ROUTER, OUT status: INTEGER)`

Output List Type: laf

Input List Type: ac

stm_r_laf_from_source_ac	**Query:** A-flow-lines whose source is a given activity **Purpose:** Returns local compound a-flow-lines that originate at activities in the input list **Syntax:** `stm_r_laf_from_source_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_laf_input_to_ac	**Query:** A-flow-lines input to a given activity **Purpose:** Returns all the local a-flow-lines **Syntax:** `stm_r_laf_input_to_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_laf_output_from_ac	**Query:** A-flow-lines output from a given activity within chart **Purpose:** Returns all local compound a-flow-lines that originate at (or inside) and terminate outside activities in the input list **Syntax:** `stm_r_laf_output_from_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_laf_to_target_ac	**Query:** A-flow-lines whose target is a given activity within chart **Purpose:** Returns local a-flow-lines (those within charts) that terminate at activities in the input list **Syntax:** `stm_r_laf_to_target_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`

Input List Type: af

stm_r_laf_contained_in_af	**Query:** None **Purpose:** Returns the local a-flow-lines that contain the global a-flow-lines in the input list **Syntax:** `stm_r_laf_contained_in_af (IN af_list: LIST OF A_FLOW_LINE, OUT status: INTEGER)`

Input List Type: ds

stm_r_laf_from_source_ds	**Query:** A-flow-lines whose source is a given data-store within chart **Purpose:** Returns local compound a-flow-lines that originate at data-stores in the input list **Syntax:** `stm_r_laf_from_source_ds (IN ds_list: LIST OF DATA_STORE, OUT status: INTEGER)`
stm_r_laf_to_target_ds	**Query:** A-flow-lines whose target is a given data-store within chart **Purpose:** Returns local compound a-flow-lines that terminate at data-stores in the input list **Syntax:** `stm_r_laf_to_target_ds (IN ds_list: LIST OF DATA_STORE, OUT status: INTEGER)`

Input List Type: mx

stm_r_laf_from_source_mx	**Query:** A-flow-lines whose source is a given element within chart **Purpose:** Returns local compound a-flow-lines whose source is an element from the input list **Syntax:** `stm_r_laf_from_source_mx (IN mx_list: LIST OF ELEMENT,OUT status: INTEGER)`
stm_r_laf_to_target_mx	**Query:** A-flow-lines whose target is given element within chart **Purpose:** Returns local compound a-flow-lines whose target is an element from the input list **Syntax:** `stm_r_laf_to_target_mx (IN mx_list: LIST OF ELEMENT,OUT status: INTEGER)`

Input List Type: router

stm_r_laf_from_source_router	**Query:** A-flow-lines whose source is a given router within chart **Purpose:** Returns local compound a-flow-lines whose source is a router from the input list **Syntax:** `stm_r_laf_from_source_router (IN router_list: LIST OF ROUTER, OUT status: INTEGER)`
stm_r_laf_to_target_router	**Query:** A-flow-lines whose target is given router within chart **Purpose:** Returns local compound a-flow-lines whose target is a router from the input list **Syntax:** `stm_r_laf_to_target_router (IN router_list: LIST OF ROUTER, OUT status: INTEGER)`

Actions (an)

This section documents the query functions that return a list of actions.

Input List Type: an

stm_r_an_by_attributes_an	**Query:** Actions by attribute **Purpose:** Returns the actions in the input list that match a given attribute and value **Syntax:** `stm_r_an_by_attributes_an (IN an_list: LIST OF ACTION, IN attr_name : STRING, IN attr_value : STRING, OUT status: INTEGER)`
stm_r_an_explicit_defined_an	**Query:** Actions explicitly defined **Purpose:** Returns the actions of the input list that were explicitly defined **Syntax:** `stm_r_an_explicit_defined_an (IN an_list: LIST OF ACTION, OUT status: INTEGER)`
stm_r_an_imp_best_match_an	**Query:** Actions whose selected implementation is **Best Match** **Purpose:** Returns the actions in the input list implemented as the **Best Match** using **Select Implementation** in the properties **Syntax:** `stm_r_an_imp_best_match_an (IN el_list: LIST OF ACTION, OUT status: INTEGER)`

stm_r_an_imp_definition_an	**Query:** Actions with a defined implementation **Purpose:** Returns the actions in the input list that have a defined implementation in the properties **Syntax:** `stm_r_an_imp_definition_an (IN el_list: LIST OF ACTION, OUT status: INTEGER)`
stm_r_an_imp_none_an	**Query:** Actions whose selected implementation is **None** **Purpose:** Returns the actions in the input list that are not implemented using Select Implementation **Syntax:** `stm_r_an_imp_none_an (IN el_list: LIST OF ACTION, OUT status: INTEGER)`
stm_r_an_imp_truth_table_an	**Query:** Actions implemented in a truth table **Purpose:** Returns the actions in the input list that are implemented with a truth table in the properties **Syntax:** `stm_r_an_imp_truth_table_an (IN el_list: LIST OF ACTION, OUT status: INTEGER)`
stm_r_an_name_of_an	**Query:** Actions whose names match a given pattern **Purpose:** Returns all the actions whose names match a specified pattern **Syntax:** `stm_r_an_name_of_an (IN pattern: STRING, OUT status: INTEGER)`
stm_r_an_synonym_of_an	**Query:** Actions whose synonyms match a given pattern **Purpose:** Returns all the actions whose synonyms match a specified pattern **Syntax:** `stm_r_an_synonym_of_an (IN pattern: STRING, OUT status: INTEGER)`
stm_r_an_unresolved_an	**Query:** Unresolved actions **Purpose:** Returns the unresolved actions in the input list **Syntax:** `stm_r_an_unresolved_an (IN an_list: LIST OF ACTION, OUT status: INTEGER)`

Input List Type: ch

stm_r_an_def_or_unres_in_ch	**Query:** Actions defined or unresolved in a given chart **Purpose:** Returns the actions that are explicitly defined or unresolved in the charts of the input list **Syntax:** `stm_r_an_def_or_unres_in_ch (IN ch_list: LIST OF CHART,OUT status: INTEGER)`
stm_r_an_defined_in_ch	**Query:** Actions defined in a given chart **Purpose:** Returns the actions that are explicitly defined in the charts of the input list **Syntax:** `stm_r_an_defined_in_ch (IN ch_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_an_unresolved_in_ch	**Query:** Actions unresolved in a given chart **Purpose:** Returns the actions that are unresolved in the charts of the input list **Syntax:** `stm_r_an_unresolved_in_ch (IN ch_list: LIST OF CHART,OUT status: INTEGER)`

Charts (ch)

This section documents the query functions that return a list of charts.

Input List Type: ac

stm_r_ch_define_ac	**Query:** Charts in which a given activity is defined **Purpose:** Returns the charts in which the activities in the input list are explicitly defined or unresolved **Syntax:** `stm_r_ch_define_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_ch_defining_ac	**Query:** Activity-charts defining a given activity **Purpose:** Returns the activity-charts that define the instance activities in the input list **Syntax:** `stm_r_ch_defining_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_ch_describing_ac	**Query:** Statecharts describing a given control activity **Purpose:** Returns the statecharts that describe the control activities in the input list **Syntax:** `stm_r_ch_describing_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`

Input List Type: an

stm_r_ch_define_an	**Query:** Charts in which a given action is defined **Purpose:** Returns the charts in which the actions in the input list are explicitly defined or unresolved **Syntax:** `stm_r_ch_define_an (IN an_list: LIST OF ACTION, OUT status: INTEGER)`

Input List Type: ch

stm_r_ch_activitychart_ch	**Query:** Activity-charts **Purpose:** Returns the activity-charts in the input list **Syntax:** `stm_r_ch_activitychart_ch (IN ch_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_ch_ancestors_of_ch	**Query:** Ancestors of a given chart **Purpose:** Returns the ancestors (in the static structure) of the charts in the input list **Syntax:** `stm_r_ch_ancestors_of_ch (IN ch_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_ch_by_attributes_ch	**Query:** Chart by attribute **Purpose:** Returns the charts in the input list that match the specified attribute name and value **Syntax:** `stm_r_ch_by_attributes_ch (IN ch_list: LIST OF CHART, IN attr_name : STRING, IN attr_value : STRING, OUT status: INTEGER)`
stm_r_ch_descendants_of_ch	**Query:** Descendants of a given chart **Purpose:** Returns the descendants (in the static structure) of the charts in the input list **Syntax:** `stm_r_ch_descendants_of_ch (IN ch_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_ch_dictionary_ch	**Query:** Global definition sets (GDSs) **Purpose:** Returns the GDSs in the input list **Syntax:** `stm_r_ch_dictionary_ch (IN el_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_ch_explicit_defined_ch	**Query:** Charts explicitly defined **Purpose:** Returns the charts of the input list that were explicitly defined **Syntax:** `stm_r_ch_explicit_defined_ch (IN ch_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_ch_generic_ch	**Query:** Generic charts **Purpose:** Returns the generic charts in the input list **Syntax:** `stm_r_ch_generic_ch (IN ch_list: LIST OF CHART,OUT status: INTEGER)`

stm_r_ch_modulechart_ch	**Query:** Module-charts **Purpose:** Returns the charts in the input list that are module-charts **Syntax:** `stm_r_ch_modulechart_ch (IN ch_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_ch_name_of_ch	**Query:** Charts whose names match a given pattern **Purpose:** Returns all the charts whose names match the specified pattern **Syntax:** `stm_r_ch_name_of_ch (IN pattern: STRING, OUT status: INTEGER)`
stm_r_ch_offpage_ch	**Query:** Offpage charts **Purpose:** Returns the offpage charts in the input list **Syntax:** `stm_r_ch_offpage_ch (IN ch_list: LIST OF CHART,OUT status: INTEGER)`
stm_r_ch_parent_ch	**Query:** Returns the parent charts of a given chart **Purpose:** Returns the parents (in the static structure) of the charts in the input list **Syntax:** `stm_r_ch_parent_ch (IN ch_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_ch_procedural_sch_ch	**Query:** Procedural statecharts **Purpose:** Returns the charts in the input list that are procedural statecharts **Syntax:** `stm_r_ch_procedural_sch_ch (IN el_list : LIST OF CHART, OUT status:INTEGER)`
stm_r_ch_referenced_all_by_ch	**Query:** Charts referenced in all levels by a given chart **Purpose:** Returns all charts referenced (instantiated) by all levels of charts in the input list **Syntax:** `stm_r_ch_referenced_all_by_ch (IN el_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_ch_referenced_by_ch	**Query:** Charts referenced by a given chart **Purpose:** Returns all charts referenced (instantiated) by the charts in the input list **Syntax:** `stm_r_ch_referenced_by_ch (IN el_list: LIST OF CHART, OUT status: INTEGER)`

stm_r_ch_root_ch	**Query:** Root charts **Purpose:** Returns the root-level charts (that have no parent) in the input list **Syntax:** `stm_r_ch_root_ch (IN ch_list: LIST OF CHART,OUT status: INTEGER)`
stm_r_ch_statechart_ch	**Query:** Statecharts **Purpose:** Returns the charts in the input list that are statecharts **Syntax:** `stm_r_ch_statechart_ch (IN ch_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_ch_subchart_ch	**Query:** Subchart of the specified chart **Purpose:** Returns the subcharts (in the static structure) of the charts in the input list **Syntax:** `stm_r_ch_subchart_ch (IN ch_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_ch_unresolved_ch	**Query:** Unresolved charts **Purpose:** Returns the unresolved charts (used but not defined) in the input list **Syntax:** `stm_r_ch_unresolved_ch (IN ch_list: LIST OF CHART, OUT status: INTEGER)`

Input List Type: co

stm_r_ch_define_co	**Query:** Charts in which a given condition is defined **Purpose:** Returns the charts in which the conditions in the input list are explicitly defined or unresolved **Syntax:** `stm_r_ch_define_co (IN co_list: LIST OF CONDITION, OUT status: INTEGER)`

Input List Type: di

stm_r_ch_define_di	**Query:** Charts in which a given data-item is defined **Purpose:** Returns the charts in which the data-items in the input list are explicitly defined or unresolved **Syntax:** `stm_r_ch_define_di (IN di_list: LIST OF DATA_ITEM, OUT status: INTEGER)`

Input List Type: ds

stm_r_ch_define_ds	**Query:** Charts in which a given data-store is defined **Purpose:** Returns the charts in which the data-stores in the input list are explicitly defined or unresolved **Syntax:** `stm_r_ch_define_ds (IN ds_list: LIST OF DATA_STORE, OUT status: INTEGER)`

Input List Type: dt

stm_r_ch_define_dt	**Query:** Charts and GDSs in which a given user-defined type is defined **Purpose:** Returns the charts in which the user-defined types in the input list are explicitly defined or unresolved **Syntax:** `stm_r_ch_define_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`

Input List Type: ev

stm_r_ch_define_ev	**Query:** Charts in which a given event is defined **Purpose:** Returns the charts in which the events in the input list are explicitly defined or unresolved **Syntax:** `stm_r_ch_define_ev (IN ev_list: LIST OF EVENT, OUT status: INTEGER)`

Input List Type: fd

stm_r_ch_define_fd	**Query:** Charts and GDSs in which a given field is defined **Purpose:** Returns the charts in which the fields in the input list are defined (in a structured data-item or user-defined type) **Syntax:** `stm_r_ch_define_fd (IN el_list: LIST OF FIELD, OUT status: INTEGER)`

Input List Type: if

stm_r_ch_define_if	**Query:** Charts in which a given information-flow is defined **Purpose:** Returns the charts in which the information-flows in the input list are explicitly defined or unresolved **Syntax:** `stm_r_ch_define_if (IN if_list: LIST OF INFORMATION_FLOW, OUT status: INTEGER)`

Input List Type: md

stm_r_ch_define_md	**Query:** Charts in which a given module is defined **Purpose:** Returns charts in which the modules in the input list are explicitly defined or unresolved **Syntax:** `stm_r_ch_define_md (IN md_list: LIST OF MODULE, OUT status: INTEGER)`
stm_r_ch_defining_md	**Query:** Module-charts defining a given module **Purpose:** Returns the module-charts that define the instance modules in the input list **Syntax:** `stm_r_ch_defining_md (IN md_list: LIST OF MODULE, OUT status: INTEGER)`
stm_r_ch_describing_md	**Query:** Activity-charts describing a given module **Purpose:** Returns the activity-charts that describe the modules in the input list **Syntax:** `stm_r_ch_describing_md (IN md_list: LIST OF MODULE, OUT status: INTEGER)`

Input List Type: mx

stm_r_ch_define_mx	**Query:** Charts in which a given element is defined **Purpose:** Returns the charts in which the elements in the input list are explicitly defined **Syntax:** `stm_r_ch_define_mx (IN mx_list: LIST OF ELEMENT, OUT status: INTEGER)`
stm_r_ch_defining_mx	**Query:** Charts defining a given element **Purpose:** Returns the charts that define the elements in the input list **Syntax:** `stm_r_ch_defining_mx (IN mx_list: LIST OF ELEMENT, OUT status: INTEGER)`
stm_r_ch_describing_mx	**Query:** Statecharts describing a given control activity **Purpose:** Returns the charts that describe the elements in the input list **Syntax:** `stm_r_ch_describing_mx (IN el_list: LIST OF ELEMENT, OUT status: INTEGER)`

Input List Type: router

stm_r_ch_define_router	**Query:** Charts in which a given router is defined **Purpose:** Returns the charts in which the routers in the input list are explicitly defined or unresolved **Syntax:** `stm_r_ch_define_router (IN router_list: LIST OF ROUTER, OUT status: INTEGER)`

Input List Type: sb

stm_r_ch_connected_to_sb	**Query:** Charts connected to a given subroutine **Purpose:** Returns the procedural Statecharts that are connected to the subroutines in the input list **Syntax:** `stm_r_ch_connected_to_sb (IN el_list: LIST OF SUBROUTINE, OUT status: INTEGER)`
stm_r_ch_define_sb	**Query:** Charts in which a given subroutine is defined **Purpose:** Returns the charts in which the subroutines in the input list are explicitly defined or unresolved **Syntax:** `stm_r_ch_define_sb (IN el_list: LIST OF SUBROUTINE, OUT status: INTEGER)`

Input List Type: st

stm_r_ch_define_st	`Query: Charts in which a given state is defined`
	`Purpose: Returns the charts in which the states in the input list are explicitly defined or unresolved`
	`Syntax:`
	`stm_r_ch_define_st (IN st_list: LIST OF STATE, OUT status: INTEGER)`

stm_r_ch_defining_st	`Query: Statecharts defining a given state`
	`Purpose: Returns the statecharts that define the instance states in the input list`
	`Syntax:`
	`stm_r_ch_defining_st (IN st_list: LIST OF STATE, OUT status: INTEGER)`

Connectors (cn)

This section documents the queries that return a list of connectors.

Input List Type: cn

stm_r_cn_deep_history_cn	**Query:** Deep history connectors **Purpose:** Returns all the deep history connectors in the input list **Syntax:** `stm_r_cn_deep_history_cn (IN cn_list: LIST OF CONNECTOR,OUT status: INTEGER)`
stm_r_cn_history_cn	**Query:** History connectors **Purpose:** Returns all the history connectors in the input list **Syntax:** `stm_r_cn_history_cn (IN cn_list: LIST OF CONNECTOR,OUT status: INTEGER)`
stm_r_cn_termination_cn	**Query:** Termination connectors **Purpose:** Returns all the termination connectors in the input list **Syntax:** `stm_r_cn_termination_cn (IN cn_list: LIST OF CONNECTOR,OUT status: INTEGER)`
stm_r_cn_source_of_bm	**Query:** Source connectors **Purpose:** Returns all the source connectors in the input list **Syntax:** `stm_list stm_r_cn_source_of_bm(stm_list in_list, int *status);`
stm_r_cn_target_of_bm	**Query:** Target connectors **Purpose:** Returns all the target connectors in the input list **Syntax:** `stm_list stm_r_cn_target_of_bm(stm_list in_list, int *status);`
stm_r_cn_source_of_bt	**Query:** Source connectors **Purpose:** Returns all the source connectors in the input list **Syntax:** `STM_R_CN_SOURCE_OF_BT(IN bt_list: LIST OF TRANSITION,OUT status: INTEGER): LIST OF CONNECTOR;`

stm_r_cn_target_of_bt	**Query:** Target connectors **Purpose:** Returns all the target connectors in the input list **Syntax:** `STM_R_CN_TARGET_OF_BT(IN bt_list: LIST OF TRANSITION,OUT status: INTEGER): LIST OF CONNECTOR;`
stm_r_cn_source_of_ba	**Query:** source connectors **Purpose:** Returns all the source connectors in the input list **Syntax:** `stm_r_cn_termination_cn (IN cn_list: LIST OF CONNECTOR,OUT status: INTEGER)`
stm_r_cn_target_of_ba	**Query:** Termination connectors **Purpose:** Returns all the history connectors in the input list **Syntax:** `STM_R_CN_TARGET_OF_BA(IN ba_list: LIST OF A_FLOW_LINE,OUT status: INTEGER): LIST OF CONNECTOR;`

Input List Type: st

stm_r_cn_history_or_term_in_st	**Query:** Termination or history connectors in a given state **Purpose:** Returns the termination and history connectors contained in the states in the input list **Syntax:** `stm_r_cn_history_or_term_in_st (IN st_list: LIST OF STATE,OUT status: INTEGER)`

Input List Type:bm

stm_r_mf_containing_bm	**Query**: M-Flow-lines **Purpose:** Returns the M-flow-lines containing the basic m-flow-lines in the input list **Syntax:** `STM_R_MF_CONTAINING_BM(IN bm_list: LIST OF M_FLOW_LINE, OUT status: INTEGER):LIST OF M_FLOW_LINE`

Input List Type: tr

stm_r_cn_source_of_tr	**Query:** History connectors sources of a given transition **Purpose:** Returns the history connectors that are sources of transitions in the input list **Syntax:** `stm_r_cn_source_of_tr (stm_list in_list, int *status);`
stm_r_cn_target_of_tr	**Query:** Termination or history connectors targets of a given transition **Purpose:** Returns the termination and history connectors that are targets of transitions in the input list **Syntax:** `stm_r_cn_target_of_tr (IN tr_list: LIST OF TRANSITION,OUT status: INTEGER)`

Conditions (co)

This section documents the query functions that return a list of conditions.

Input List Type: af

stm_r_co_flowing_through_af	**Query:** Conditions flowing through a given a-flow-line **Purpose:** Returns the conditions actually flowing through a-flow-lines in the input list **Syntax:** `stm_r_co_flowing_through_ af (IN af_list: LIST OF A_FLOW_LINE, OUT status: INTEGER)`
stm_r_co_labeling_af	**Query:** Conditions labeling a given a-flow-line **Purpose:** Returns the conditions which label the a-flow-lines in the input list **Syntax:** `stm_r_co_labeling_af (IN af_list: LIST OF A_FLOW_LINE, OUT status: INTEGER)`

Input List Type: ch

stm_r_co_def_or_unres_in_ch	**Query:** Conditions defined or unresolved in a given chart **Purpose:** Returns conditions that are explicitly defined or unresolved in the charts of the input list **Syntax:** `stm_r_co_def_or_unres_in_ch (IN ch_list: LIST OF CHART,OUT status: INTEGER)`
stm_r_co_defined_in_ch	**Query:** Conditions defined in a given chart **Purpose:** Returns the conditions that are explicitly defined in the charts of the input list **Syntax:** `stm_r_co_defined_in_ch (IN ch_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_co_unresolved_in_ch	**Query:** Conditions unresolved in a given chart **Purpose:** Returns conditions that are unresolved in the charts of the input list **Syntax:** `stm_r_co_unresolved_in_ch (IN ch_list: LIST OF CHART,OUT status: INTEGER)`

Input List Type: co

stm_r_co_array_co	**Query:** Conditions by subtype **Purpose:** Returns the conditions in the input list that are defined as array **Syntax:** `stm_r_co_array_co (IN co_list: LIST OF CONDITION, OUT status INTEGER)`
stm_r_co_by_attributes_co	**Query:** Conditions by attributes **Purpose:** Returns the conditions in the input list that match the specified attribute name and value **Syntax:** `stm_r_co_by_attributes_co (IN co_list: LIST OF CONDITION, IN attr_name : STRING, IN attr_value : STRING, OUT status: INTEGER)`

stm_r_co_by_structure_type_co	**Query:** None **Purpose:** Returns the conditions in the input list that have the specified structure type (for example, single or array) **Syntax:** `stm_r_co_by_structure_type_co (IN co_list: LIST OF CONDITION, IN dtype: INTEGER, OUT status: INTEGER)`
stm_r_co_callback_binding_co	**Query:** Conditions with callback bindings **Purpose:** Returns the conditions of the input list that have callback bindings **Syntax:** `stm_r_co_callback_binding_co (IN el_list: LIST OF CONDITION, OUT status: INTEGER)`
stm_r_co_explicit_defined_co	**Query:** Conditions explicitly defined **Purpose:** Returns the conditions of the input list that were explicitly defined **Syntax:** `stm_r_co_explicit_defined_co (IN co_list: LIST OF CONDITION, OUT status: INTEGER)`
stm_r_co_name_of_co	**Query:** Conditions whose names match a given pattern **Purpose:** Returns all the conditions whose names match the specified pattern **Syntax:** `stm_r_co_name_of_co (IN pattern: STRING, OUT status: INTEGER)`
stm_r_co_single_co	**Query:** Conditions by subtype **Purpose:** Returns the conditions in the input list that are defined as single **Syntax:** `stm_r_co_single_co (IN co_list: LIST OF CONDITION, OUT status: INTEGER)`
stm_r_co_synonym_of_co	**Query:** Conditions whose synonyms match a given pattern **Purpose:** Returns all the conditions whose synonyms match the specified pattern **Syntax:** `stm_r_co_synonym_of_co (IN pattern: STRING, OUT status: INTEGER)`
stm_r_co_unresolved_co	**Query:** Unresolved conditions **Purpose:** Returns the unresolved conditions in the input list **Syntax:** `stm_r_co_unresolved_co (IN co_list: LIST OF CONDITION, OUT status: INTEGER)`

Input List Type: di

stm_r_co_contained_in_di	**Query:** Conditions contained in a given data-item **Purpose:** Returns the conditions contained in data-items from the input list (conditions appearing in the **Consists of** field of a data-item) **Syntax:** `stm_r_co_contained_in_di (IN di_list: LIST OF DATA_ITEM, OUT status: INTEGER)`

Input List Type: if

stm_r_co_contained_in_if	**Query:** Conditions contained in a given information-flow **Purpose:** Returns the conditions contained in information-flows from the input list (conditions appearing in the **Consists of** field of an information-flow) **Syntax:** `stm_r_co_contained_in_if (IN if_list: LIST OF INFORMATION_FLOW, OUT status: INTEGER)`

Input List Type: mf

stm_r_co_flowing_through_mf	**Query:** Conditions flowing through a given m-flow-line **Purpose:** Returns the conditions actually flowing through m-flow-lines in the input list **Syntax:** `stm_r_co_flowing_through_ mf (IN mf_list: LIST OF M_FLOW_LINE, OUT status: INTEGER)`
stm_r_co_labeling_mf	**Query:** Conditions labeling a given m-flow-line **Purpose:** Returns the conditions that label the m-flow-lines in the input list **Syntax:** `stm_r_co_labeling_mf (IN mf_list: LIST OF M_FLOW_LINE, OUT status: INTEGER)`

Data-Items (di)

This section documents the query functions that return a list of data-items.

Input List Type: af

stm_r_di_flowing_through_af	**Query:** Data-items flowing through a given a-flow-line **Purpose:** Returns the data-items actually flowing through a-flow-lines in the input list **Syntax:** `stm_r_di_flowing_through_ af (IN af_list: LIST OF A_FLOW_LINE, OUT status: INTEGER)`
stm_r_di_labeling_af	**Query:** Data-items labeling a given a-flow-line **Purpose:** Returns the data-items which label the a-flow-lines in the input list **Syntax:** `stm_r_di_labeling_af (IN af_list: LIST OF A_FLOW_LINE, OUT status: INTEGER)`

Input List Type: ch

stm_r_di_def_or_unres_in_ch	**Query:** Data-items defined or unresolved in a given chart **Purpose:** Returns the data-items explicitly defined or unresolved in the charts of the input list **Syntax:** `stm_r_di_def_or_unres_in_ch (IN ch_list: LIST OF CHART,OUT status: INTEGER)`
stm_r_di_defined_in_ch	**Query:** Data-items defined in a given chart **Purpose:** Returns the data-items explicitly defined in the charts of the input list **Syntax:** `stm_r_di_defined_in_ch (IN ch_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_di_unresolved_in_ch	**Query:** Data-items unresolved in a given chart **Purpose:** Returns the data-items that are unresolved in the charts of the input list **Syntax:** `stm_r_di_unresolved_in_ch (IN ch_list: LIST OF CHART,OUT status: INTEGER)`

Input List Type: co

stm_r_di_containing_co	**Query:** Data-item containing a given condition **Purpose:** Returns the data-items containing the conditions in the input list (as defined in the **Consists of** field of the data-item's form) **Syntax:** `stm_r_di_containing_co (IN co_list: LIST OF CONDITION, OUT status: INTEGER)`

Input List Type: di

stm_r_di_array_di	**Query:** Data-items by subtype **Purpose:** Returns the data-items in the input list that are defined as array **Syntax:** `stm_r_di_array_di (IN di_list: LIST OF DATA_ITEM, OUT status: INTEGER)`
stm_r_di_array_missing_di	**Query:** Array of data-items by subtype **Purpose:** Returns the arrays of data-items in the input list for which no type is defined **Syntax:** `stm_r_di_array_missing_di (IN di_list: LIST OF DATA_ITEM, OUT status: INTEGER)`
stm_r_di_basic_di	**Query:** Basic data-items **Purpose:** Returns the data-items in the input list that are basic (not defined using other data-items) **Syntax:** `stm_r_di_basic_di (IN di_list: LIST OF DATA_ITEM, OUT status: INTEGER)`
stm_r_di_bit_di	**Query:** Basic data-items **Purpose:** Returns the data-items in the input list that are defined as Bit in the **Structure/Type** field of the data-item form **Syntax:** `stm_r_di_bit_di (IN di_list: LIST OF DATA_ITEM, OUT status: INTEGER)`

stm_r_di_bit_queue_di	**Query:** Data-items by subtype **Purpose:** Returns the data-items in the input list that are defined as queue of bits **Syntax:** `stm_r_di_bit_queue_di (IN el_list: LIST OF DATA_ITEM, OUT status: INTEGER)`
stm_r_di_bits_array_di	**Query:** Data-items by subtype **Purpose:** Returns the data-items in the input list that are defined as array of bit array **Syntax:** `stm_r_di_bits_array_di (IN di_list: LIST OF DATA_ITEM, OUT status: INTEGER)`
stm_r_di_bits_di	**Query:** Data-items by subtype **Purpose:** Returns the data-items in the input list that are defined as bit-array in the **Structure/Type** field of the data-item form **Syntax:** `stm_r_di_bits_di (IN di_list: LIST OF DATA_ITEM, OUT status: INTEGER)`
stm_r_di_bits_queue_di	**Query:** Data-items by subtype **Purpose:** Returns the data-items in the input list that are defined as queue of bit array **Syntax:** `stm_r_di_bits_queue_di (IN el_list: LIST OF DATA_ITEM, OUT status: INTEGER)`
stm_r_di_by_attributes_di	**Query:** Data-items by attributes **Purpose:** Returns the data-items in the input list that match the specified attribute name and value **Syntax:** `stm_r_di_by_attributes_di (IN di_list: LIST OF DATA_ITEM, IN attr_name : STRING, IN attr_value : STRING, OUT status: INTEGER)`
stm_r_di_by_structure_type_di	**Query:** None **Purpose:** Returns the data-items in the input list that have a particular structure type (for example, single, array, or queue) **Syntax:** `stm_r_di_by_structure_type_di (IN di_list: LIST OF DATA_ITEM, IN dtype: INTEGER, OUT status: INTEGER)`
stm_r_di_callback_binding_di	**Query:** Data-items with callback bindings **Purpose:** Returns the data-items of the input list that have callback bindings **Syntax:** `stm_r_di_callback_binding_di (IN el_list: LIST OF DATA_ITEM, OUT status: INTEGER)`

stm_r_di_explicit_defined_di	**Query:** Data-items explicitly defined **Purpose:** Returns the data-items of the input list that were explicitly defined **Syntax:** `stm_r_di_explicit_defined_di (IN di_list: LIST OF DATA_ITEM, OUT status: INTEGER)`
stm_r_di_integer_di	**Query:** Integer subtype **Purpose:** Returns the data-items in the input list that are defined as integer in the **Structure/Type** field of the data-item's form **Syntax:** `stm_r_di_integer_di (IN di_list: LIST OF DATA_ITEM, OUT status: INTEGER)`
stm_r_di_integer_array_di	**Query:** Data-items by subtype **Purpose:** Returns the data-items in the input list that are defined as array of integer **Syntax:** `stm_r_di_integer_array_di (IN di_list: LIST OF DATA_ITEM, OUT status: INTEGER)`
stm_r_di_integer_queue_di	**Query:** Data-items by subtype **Purpose:** Returns the data-items in the input list that are defined as queue of integer **Syntax:** `stm_r_di_integer_queue_di (IN el_list: LIST OF DATA_ITEM, OUT status: INTEGER)`
stm_r_di_missing_di	**Query:** Data-item by subtype **Purpose:** Returns the data-items in the input list for which no type is defined **Syntax:** `stm_r_di_missing_di (IN el_list: LIST OF DATA_ITEM, OUT status: INTEGER)`
stm_r_di_name_of_di	**Query:** Data-items whose names match a given pattern **Purpose:** Returns all the data-items whose names match the specified pattern **Syntax:** `stm_r_di_name_of_di (IN pattern: STRING, OUT status: INTEGER)`
stm_r_di_parent_of_di	**Query:** Parent data-items of a given data-item **Purpose:** Returns the data-items containing the data-items from the input list (as defined in the **Consists of** field of the data-item's form) **Syntax:** `stm_r_di_parent_of_di (IN di_list: LIST OF DATA_ITEM, OUT status: INTEGER)`

stm_r_di_queue_di	**Query:** Data-items by subtype **Purpose:** Returns the data-items in the input list that are defined as queue **Syntax:** `stm_r_di_queue_di (IN el_list: LIST OF DATA_ITEM, OUT status: INTEGER)`
stm_r_di_queue_missing_di	**Query:** Queues of data-items by subtype **Purpose:** Returns the queues of data-items in the input list for which no type is defined **Syntax:** `stm_r_di_queue_missing_di (IN el_list: LIST OF DATA_ITEM, OUT status: INTEGER)`
stm_r_di_real_di	**Query:** Real subtype **Purpose:** Returns the data-items from the input list that are defined as Real (Float) in the **Structure/Type** field of the data-item's form **Syntax:** `stm_r_di_real_di (IN di_list: LIST OF DATA_ITEM, OUT status: INTEGER)`
stm_r_di_real_array_di	**Query:** Data-items by subtype **Purpose:** Returns the data-items in the input list that are defined as real **Syntax:** `stm_r_di_real_array_di (IN di_list: LIST OF DATA_ITEM, OUT status: INTEGER)`
stm_r_di_real_queue_di	**Query:** Data-items by subtype **Purpose:** Returns the data-items in the input list that are defined as queue of real **Syntax:** `stm_r_di_real_queue_di (IN el_list: LIST OF DATA_ITEM, OUT status: INTEGER)`
stm_r_di_record_array_di	**Query:** Data-items by subtype **Purpose:** Returns the data-items in the input list that are defined as array of record **Syntax:** `stm_r_di_record_array_di (IN el_list: LIST OF DATA_ITEM, OUT status: INTEGER)`
stm_r_di_record_di	**Query:** Record subtype **Purpose:** Returns the data-items from the input list that are defined as Record in the **Structure/Type** field of the data-item's form **Syntax:** `stm_r_di_record_di (IN di_list: LIST OF DATA_ITEM, OUT status: INTEGER)`

stm_r_di_single_di	**Query:** Data-items by subtype **Purpose:** Returns the data-items in the input list that are defined as single **Syntax:** `stm_r_di_single_di (IN di_list: LIST OF DATA_ITEM, OUT status: INTEGER)`
stm_r_di_string_array_di	**Query:** Data-items by subtype **Purpose:** Returns the data-items in the input list that are defined as array of string **Syntax:** `stm_r_di_string_array_di (IN di_list: LIST OF DATA_ITEM, OUT status: INTEGER)`
stm_r_di_string_di	**Query:** String subtype **Purpose:** Returns the data-items from the input list that are defined as String in the **Structure/Type** field of the data-item's form **Syntax:** `stm_r_di_string_di (IN di_list: LIST OF DATA_ITEM, OUT status: INTEGER)`
stm_r_di_string_queue_di	**Query:** Data-items by subtype **Purpose:** Returns the data-items in the input list that are defined as queue of string **Syntax:** `stm_r_di_string_queue_di (IN el_list: LIST OF DATA_ITEM, OUT status: INTEGER)`
stm_r_di_subdata_item_of_di	**Query:** Subdata-item of a given data-item **Purpose:** Returns the data-items that are components of data-items in the input list (as defined in the **Consists of** field of the data-item's form) **Syntax:** `stm_r_di_subdata_item_of_di (IN di_list: LIST OF DATA_ITEM, OUT status: INTEGER)`
stm_r_di_synonym_of_di	**Query:** Data-items whose synonyms match a given pattern **Purpose:** Returns all the data-items whose synonyms match the specified pattern **Syntax:** `stm_r_di_synonym_of_di (IN pattern: STRING, OUT status: INTEGER)`
stm_r_di_union_array_di	**Query:** Data-items by subtype **Purpose:** Returns the data-items in the input list that are defined as array **Syntax:** `stm_r_di_union_array_di (IN el_list: LIST OF DATA_ITEM, OUT status: INTEGER)`

stm_r_di_union_di	**Query:** Data-items by subtype **Purpose:** Returns the data-items in the input list that are defined as union **Syntax:** `stm_r_di_union_di (IN el_list: LIST OF DATA_ITEM, OUT status: INTEGER)`
stm_r_di_unresolved_di	**Query:** Unresolved data-items **Purpose:** Returns the unresolved data-items in the input list **Syntax:** `stm_r_di_unresolved_di (IN di_list: LIST OF DATA_ITEM, OUT status: INTEGER)`
stm_r_di_user_type_di	**Query:** Data-items by subtype **Purpose:** Returns the data-items in the input list that are defined as user-defined type **Syntax:** `stm_r_di_user_type_di (IN el_list: LIST OF DATA_ITEM, OUT status: INTEGER)`
stm_r_di_user_type_array_di	**Query:** Data-items by subtype **Purpose:** Returns the data-items in the input list that are defined as array of user-defined type **Syntax:** `stm_r_di_user_type_array_di (IN el_list: LIST OF DATA_ITEM, OUT status: INTEGER)`
stm_r_di_user_type_queue_di	**Query:** Data-items by subtype **Purpose:** Returns the data-items in the input list that are defined as queue of user-defined type **Syntax:** `stm_r_di_user_type_queue_di (IN el_list: LIST OF DATA_ITEM, OUT status: INTEGER)`

Input List Type: fd

stm_r_di_containing_fd	**Query:** Data-items containing a given field **Purpose:** Returns the data-items (records or unions) in which the fields in the input list are defined **Syntax:** `stm_r_di_containing_fd (IN el_list: LIST OF FIELD, OUT status: INTEGER)`

Input List Type: if

stm_r_di_contained_in_if	**Query:** Data-items contained in a given information-flow **Purpose:** Returns the data-items contained in information-flow from the input list (as defined in the **Consists of** field of the information-flow's form) **Syntax:** **stm_r_di_contained_in_if (IN if_list: LIST OF INFORMATION_FLOW, OUT status: INTEGER)**

Input List Type: mf

stm_r_di_flowing_through_mf	**Query:** Data-items flowing through a given m-flow-line **Purpose:** Returns the data-items actually flowing through m-flow-lines in the input list **Syntax:** `stm_r_di_flowing_through_mf (IN mf_list: LIST OF M_FLOW_LINE, OUT status: INTEGER)`
stm_r_di_labeling_mf	**Query:** Data-items labeling a given m-flow-line **Purpose:** Returns the data-items which label the m-flow-lines in the input list **Syntax:** `stm_r_di_labeling_mf (IN mf_list: LIST OF M_FLOW_LINE, OUT status: INTEGER)`

Data-Stores (ds)

This section documents the query functions that return a list of data-stores.

Input List Type: ac

stm_r_ds_contained_in_ac	**Query:** Data-stores contained in a given activity **Purpose:** Returns the data-stores contained directly in activities from the input list **Syntax:** `stm_r_ds_contained_in_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_ds_in_ac	**Query:** Data-stores in a given activity **Purpose:** Returns the data-stores contained in the activities from the input list **Syntax:** `stm_r_ds_in_ac (IN el_list: LIST OF DATA_STORE, OUT status: INTEGER)`

Input List Type: af

stm_r_ds_source_of_af	**Query:** Data-stores that are sources of a given a-flow-line **Purpose:** Returns the data-stores that are sources of a-flow-lines in the input list **Syntax:** `stm_r_ds_source_of_af (IN af_list: LIST OF A_FLOW_LINE, OUT status: INTEGER)`
stm_r_ds_target_of_af	**Query:** Data-stores that are targets of a given a-flow-line **Purpose:** Returns the data-stores that are targets of a-flow-lines in the input list **Syntax:** `stm_r_ds_target_of_af (IN af_list: LIST OF A_FLOW_LINE, OUT status: INTEGER)`

Input List Type: ch

stm_r_ds_def_or_unres_in_ch	**Query:** Data-stores defined or unresolved in a given chart **Purpose:** Returns the data-stores that are explicitly defined or unresolved in the charts of the input list **Syntax:** `stm_r_ds_def_or_unres_in_ch (IN ch_list: LIST OF CHART,OUT status: INTEGER)`
stm_r_ds_defined_in_ch	**Query:** Data-stores defined in a given chart **Purpose:** Returns the data-stores that are explicitly defined in the charts of the input list **Syntax:** `stm_r_ds_defined_in_ch (IN ch_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_ds_unresolved_in_ch	**Query:** Data-stores unresolved in a given chart **Purpose:** Returns the data-stores that are unresolved in the charts of the input list **Syntax:** `stm_r_ds_unresolved_in_ch (IN ch_list: LIST OF CHART,OUT status: INTEGER)`

Input List Type: ds

stm_r_ds_by_attributes_ds	**Query:** Data-stores by attributes **Purpose:** Returns the data-stores in the input list that match a given attribute name and value **Syntax:** `stm_r_ds_by_attributes_ds (IN ds_list: LIST OF DATA_STORE, IN attr_name: STRING, IN attr_value: STRING, OUT status: INTEGER)`
stm_r_ds_explicit_defined_ds	**Query:** Data-stores explicitly defined **Purpose:** Returns the data-stores of the input list that were explicitly defined **Syntax:** `stm_r_ds_explicit_defined_ds (IN ds_list: LIST OF DATA_STORE, OUT status: INTEGER)`

stm_r_ds_is_occurrence_of_ds	**Query:** Data-store occurrences of a given data-store **Purpose:** Returns the data-stores for which the data-stores in the input list appear in the **Is Data-store** field of their form **Syntax:** `stm_r_ds_is_occurrence_of_ds (IN ds_list: LIST OF DATA_STORE, OUT status: INTEGER)`
stm_r_ds_is_principal_of_ds	**Query:** Principal data-stores of a given data-store **Purpose:** Returns the data-stores for which the data-stores in the input list appear in the **Is Data-store** field of their form **Syntax:** `stm_r_ds_is_principal_of_ds (IN ds_list: LIST OF DATA_STORE, OUT status: INTEGER)`
stm_r_ds_name_of_ds	**Query:** Data-store whose names match a given pattern **Purpose:** Returns all the data-stores whose names match the specified pattern **Syntax:** `stm_r_ds_name_of_ds (IN pattern: STRING, OUT status: INTEGER)`
stm_r_ds_synonym_of_ds	**Query:** Data-store whose synonyms match a given pattern **Purpose:** Returns all the data-stores whose synonyms match the specified pattern **Syntax:** `stm_r_ds_synonym_of_ds (IN pattern: STRING, OUT status: INTEGER)`
stm_r_ds_unresolved_ds	**Query:** Unresolved data-stores **Purpose:** Returns the unresolved data-stores in the input list **Syntax:** `stm_r_ds_unresolved_ds (IN ds_list: LIST OF DATA_STORE, OUT status: INTEGER)`

Input List Type: md

stm_r_ds_resides_in_md	**Query:** Data-stores residing in a given module. **Purpose:** Returns the data-stores residing in modules from the input list. The module appears in the **Resides in Module** field of the data-store's form. **Syntax:** `stm_r_ds_resides_in_md (IN md_list: LIST OF MODULE, OUT status: INTEGER)`

User-Defined Types (dt)

This section documents the query functions that return a list of data-types.

Input List Type: ch

stm_r_dt_def_or_unres_in_ch	**Query:** User-defined types defined or unresolved in a given chart **Purpose:** Returns the user-defined types that are explicitly defined or unresolved in the charts in the input list **Syntax:** `stm_r_dt_def_or_unres_in_ch (IN el_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_dt_defined_in_ch	**Query:** User-defined types defined in a given chart **Purpose:** Returns the user-defined types that are explicitly defined in the charts in the input list **Syntax:** `stm_r_dt_defined_in_ch (IN el_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_dt_unresolved_in_ch	**Query:** User-defined types unresolved in a given chart **Purpose:** Returns the user-defined types that are unresolved in the charts in the input list **Syntax:** `stm_r_dt_unresolved_in_ch (IN el_list: LIST OF CHART, OUT status: INTEGER)`

Input List Type: dt

stm_r_dt_array_dt	**Query:** User-defined types by subtype **Purpose:** Returns the user-defined types in the input list that are defined as array **Syntax:** `stm_r_dt_array_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`
stm_r_dt_array_missing_dt	**Query:** Arrays of user-defined type by subtype **Purpose:** Returns the arrays of user-defined types in the input list for which no type is defined **Syntax:** `stm_r_dt_array_missing_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`
stm_r_dt_bit_dt	**Query:** User-defined types by subtype **Purpose:** Returns the user-defined types in the input list that are defined as bit **Syntax:** `stm_r_dt_bit_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`
stm_r_dt_bit_queue_dt	**Query:** User-defined types by subtype **Purpose:** Returns the user-defined types in the input list that are defined as queue of bit **Syntax:** `stm_r_dt_bit_queue_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`
stm_r_dt_bits_array_dt	**Query:** User-defined types by subtype **Purpose:** Returns the user-defined types in the input list that are defined as array of bit array **Syntax:** `stm_r_dt_bits_array_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`
stm_r_dt_bits_dt	**Query:** User-defined types by subtype **Purpose:** Returns the user-defined types in the input list that are defined as bit array **Syntax:** `stm_r_dt_bits_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`

stm_r_dt_bits_queue_dt	**Query:** User-defined types by subtype **Purpose:** Returns the user-defined types in the input list that are defined as queue of bit array **Syntax:** `stm_r_dt_bits_queue_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`
stm_r_dt_by_attributes_dt	**Query:** User-defined types by attribute **Purpose:** Returns the user-defined types in the input list that match a given attribute and value **Syntax:** `stm_r_dt_by_attributes_dt (IN dt_list: LIST OF DATA_TYPE, IN attr_name: STRING, IN attr_value: STRING, OUT status: INTEGER)`
stm_r_dt_by_structure_type_dt	**Query:** None **Purpose:** Returns the user-defined types in the input list that have a given structure type (for example, single, array or queue) **Syntax:** `stm_r_dt_by_structure_type_dt (IN el_list: LIST OF DATA_TYPE, IN dtype: INTEGER, OUT status: INTEGER)`
stm_r_dt_condition_array_dt	**Query:** User-defined types by subtype **Purpose:** Returns the user-defined types in the input list that are defined as array of condition **Syntax:** `stm_r_dt_condition_array_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`
stm_r_dt_condition_dt	**Query:** User-defined types by subtype **Purpose:** Returns the user-defend types in the input list that are defined as condition **Syntax:** `stm_r_dt_condition_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`
stm_r_dt_condition_queue_dt	**Query:** User-defined types by subtype **Purpose:** Returns the user-defined types in the input list that are defined as queue of condition **Syntax:** `stm_r_dt_condition_queue_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`
stm_r_dt_enums_dt	**Query:** User-defined types defined as enumerated types **Purpose:** Returns the user-defined types in the input list that are defined as enumerated types **Syntax:** `stm_r_dt_enums_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`

stm_r_dt_explicit_defined_dt	**Query:** User-defined types explicitly defined **Purpose:** Returns the user-defined types in the input list that are explicitly defined **Syntax:** `stm_r_dt_explicit_defined_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`
stm_r_dt_integer_dt	**Query:** User-defined types by subtype **Purpose:** Returns the user-defined types in the input list that are defined as integer **Syntax:** `stm_r_dt_integer_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`
stm_r_dt_integer_array_dt	**Query:** User-defined types by subtype **Purpose:** Returns the user-defined types in the input list that are defined as array of integer **Syntax:** `stm_r_dt_integer_array_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`
stm_r_dt_integer_queue_dt	**Query:** User-defined types by subtype **Purpose:** Returns the user-defined types in the input list that are defined as queue of integer **Syntax:** `stm_r_dt_integer_queue_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`
stm_r_dt_missing_dt	**Query:** User-defined type by subtype **Purpose:** Returns the user-defined types in the input list for which no type is defined **Syntax:** `stm_r_dt_missing_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`
stm_r_dt_name_of_dt	**Query:** User-defined types whose names match a given pattern **Purpose:** Returns all user-defined types whose names match the specified pattern **Syntax:** `stm_r_dt_name_of_dt (IN pattern: STRING, OUT status: INTEGER)`
stm_r_dt_queue_dt	**Query:** User-defined types by subtype **Purpose:** Returns the user-defined types in the input list that are defined as queue **Syntax:** `stm_r_dt_queue_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`

stm_r_dt_queue_missing_dt	**Query:** Queues of user-defined type by subtype **Purpose:** Returns the queues of user-defined types in the input list for which no type is defined **Syntax:** `stm_r_dt_queue_missing_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`
stm_r_dt_real_array_dt	**Query:** User-defined types by subtype **Purpose:** Returns the user-defined types in the input list that are defined as real **Syntax:** `stm_r_dt_real_array_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`
stm_r_dt_real_dt	**Query:** User-defined types by subtype **Purpose:** Returns the user-defined types in the input list that are defined as real **Syntax:** `stm_r_dt_real_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`
stm_r_dt_real_queue_dt	**Query:** User-defined types by subtype **Purpose:** Returns the user-defined types in the input list that are defined as queue of real **Syntax:** `stm_r_dt_real_queue_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`
stm_r_dt_record_array_dt	**Query:** User-defined types by subtype **Purpose:** Returns the user-defined types in the input list that are defined as array of record **Syntax:** `stm_r_dt_record_array_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`
stm_r_dt_record_dt	**Query:** User-defined types by subtype **Purpose:** Returns the user-defined types in the input list that are defined as record **Syntax:** `stm_r_dt_record_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`
stm_r_dt_single_dt	**Query:** User-defined types by subtype **Purpose:** Returns the user-defined types in the input list that are defined as single **Syntax:** `stm_r_dt_single_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`

stm_r_dt_string_array_dt	**Query:** User-defined types by subtype **Purpose:** Returns the user-defined types in the input list that are defined as array of string **Syntax:** `stm_r_dt_string_array_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`
stm_r_dt_string_dt	**Query:** User-defined types by subtype **Purpose:** Returns the user-defined types in the input list that are defined as string **Syntax:** `stm_r_dt_string_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`
stm_r_dt_string_queue_dt	**Query:** User-defined types by subtype **Purpose:** Returns the user-defined types in the input list that are defined as queue of string **Syntax:** `stm_r_dt_string_queue_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`
stm_r_dt_synonym_of_dt	**Query:** User-defined types whose synonyms match a given pattern **Purpose:** Returns all user-defined types whose synonyms match the specified pattern `Syntax: stm_r_dt_synonym_of_dt (IN pattern: STRING, OUT status: INTEGER)`
stm_r_dt_union_dt	**Query:** User-defined type by subtype **Purpose:** Returns the user-defined types in the input list that are defined as union **Syntax:** `stm_r_dt_union_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`
stm_r_dt_union_array_dt	**Query:** User-defined type by subtype **Purpose:** Returns the user-defined types in the input list that are defined as array of union **Syntax:** `stm_r_dt_union_array_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`
stm_r_dt_unresolved_dt	**Query:** Unresolved user-defined types **Purpose:** Returns the unresolved user-defined types in the input list **Syntax:** `stm_r_dt_unresolved_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`

stm_r_dt_user_type_array_dt	**Query:** User-defined types by subtype **Purpose:** Returns the user-defined types in the input list that are defined as array of another user-defined type **Syntax:** `stm_r_dt_user_type_array_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`
stm_r_dt_user_type_dt	**Query:** User-defined types by subtype **Purpose:** Returns the user-defined types in the input list that are defined as other user-defined type **Syntax:** `stm_r_dt_user_type_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`
stm_r_dt_user_type_queue_dt	**Query:** User-defined types by subtype **Purpose:** Returns the user-defined types in the input list that are defined as queue of another user-defined type **Syntax:** `stm_r_dt_user_type_queue_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`

Input List Type: fd

stm_r_dt_containing_fd	**Query:** User-defined types containing a given field **Purpose:** Returns the user-defined types (records or unions), in which the fields in the input list are defined **Syntax:** `stm_r_dt_containing_fd (IN el_list: LIST OF FIELD, OUT status: INTEGER)`

Events (ev)

This section documents the query functions that return a list of events.

Input List Type: af

stm_r_ev_flowing_through_af	**Query:** Events flowing through the specified a-flow-line **Purpose:** Returns the events actually flowing through a-flow-lines in the input list **Syntax:** `stm_r_ev_flowing_through_af (IN af_list: LIST OF A_FLOW_LINE, OUT status: INTEGER)`
stm_r_ev_labeling_af	**Query:** Events labeling a given a-flow-line **Purpose:** Returns the events that label the a-flow-lines in the input list **Syntax:** `stm_r_ev_labeling_af (IN af_list: LIST OF A_FLOW_LINE, OUT status: INTEGER)`

Input List Type: ch

stm_r_ev_def_or_unres_in_ch	**Query:** Events defined or unresolved in a given chart **Purpose:** Returns the events that are explicitly defined or unresolved in the charts of the input list **Syntax:** `stm_r_ev_def_or_unres_in_ch (IN ch_list: LIST OF CHART,OUT status: INTEGER)`
stm_r_ev_defined_in_ch	**Query:** Events defined in a given chart **Purpose:** Returns the events that are explicitly defined in the charts of the input list **Syntax:** `stm_r_ev_defined_in_ch (IN ch_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_ev_unresolved_in_ch	**Query:** Events unresolved in a given chart **Purpose:** Returns the events that are unresolved in the charts of the input list **Syntax:** `stm_r_ev_unresolved_in_ch (IN ch_list: LIST OF CHART,OUT status: INTEGER)`

Input List Type: ev

stm_r_ev_array_ev	**Query:** Events by subtype **Purpose:** Returns the events in the input list that are defined as array **Syntax:** `stm_r_ev_array_ev (IN ev_list: LIST OF EVENT, OUT status: INTEGER)`
stm_r_ev_by_attributes_ev	**Query:** Events by attributes **Purpose:** Returns the events in the input list that match the specified attribute name and value **Syntax:** `stm_r_ev_by_attributes_ev (IN ev_list: LIST OF EVENT, IN attr_name: STRING, IN attr_value: STRING, OUT status: INTEGER)`
stm_r_ev_by_structure_type_ev	**Query:** None **Purpose:** Returns the events in the input list that have the specified structure type (for example, single or array) **Syntax:** `stm_r_ev_by_structure_type_ev (IN ev_list: LIST OF EVENT, IN dtype: INTEGER, OUT status: INTEGER)`
stm_r_ev_callback_binding_ev	**Query:** Events with callback bindings **Purpose:** Returns the events in the input list that have callback bindings **Syntax:** `stm_r_ev_callback_binding_ev (IN el_list: LIST OF EVENT, OUT status: INTEGER)`
stm_r_ev_explicit_defined_ev	**Query:** Events explicitly defined **Purpose:** Returns the events of the input list that were explicitly defined **Syntax:** `stm_r_ev_explicit_defined_ev (IN ev_list: LIST OF EVENT, OUT status: INTEGER)`
stm_r_ev_name_of_ev	**Query:** Events whose names match a given pattern **Purpose:** Returns all the events whose names match the specified pattern **Syntax:** `stm_r_ev_name_of_ev (IN pattern: STRING, OUT status: INTEGER)`

stm_r_ev_single_ev	**Query:** Events by subtype **Purpose:** Returns the events in the input list that are defined as single **Syntax:** `stm_r_ev_single_ev (IN ev_list: LIST OF EVENT, OUT status: INTEGER)`
stm_r_ev_synonym_of_ev	**Query:** Events whose synonyms match a given pattern **Purpose:** Returns all the events whose synonyms match the specified pattern **Syntax:** `stm_r_ev_synonym_of_ev (IN pattern: STRING, OUT status: INTEGER)`
stm_r_ev_unresolved_ev	**Query:** Unresolved events **Purpose:** Returns the unresolved events in the input list **Syntax:** `stm_r_ev_unresolved_ev (IN ev_list: LIST OF EVENT, OUT status: INTEGER)`

Input List Type: if

stm_r_ev_contained_in_if	**Query:** Events contained in a given information-flow **Purpose:** Returns the events contained in information-flows from the input list (events used in the **Consists of** field of the information-flow's form) **Syntax:** `stm_r_ev_contained_in_if (IN if_list: LIST OF INFORMATION_FLOW, OUT status: INTEGER)`

Input List Type: mf

stm_r_ev_flowing_through_mf	**Query:** Events flowing through a given m-flow-line **Purpose:** Returns the events actually flowing through m-flow-lines from the input list **Syntax:** `stm_r_ev_flowing_through_mf (IN mf_list: LIST OF M_FLOW_LINE, OUT status: INTEGER)`
stm_r_ev_labeling_mf	**Query:** Events labeling a given m-flow-line **Purpose:** Returns the events that label the m-flow-lines in the input list **Syntax:** `stm_r_ev_labeling_mf (IN mf_list: LIST OF M_FLOW_LINE, OUT status: INTEGER)`

Fields (fd)

This section documents the queries that return a list of fields.

Input List Type: ch

stm_r_fd_defined_in_ch	**Query:** Fields defined in a given chart **Purpose:** Returns the fields that are part of the structured data-items in the input list **Syntax:** `stm_r_fd_defined_in_ch (IN el_list: LIST OF CHART, OUT status: INTEGER)`

Input List Type: di

stm_r_fd_contained_in_di	**Query:** Fields by subtype **Purpose:** Returns the fields in the input list that are defined as array **Syntax:** `stm_r_fd_contained_in_di (IN el_list: LIST OF DATA_ITEM, OUT status: INTEGER)`

Input List Type: dt

stm_r_fd_contained_in_dt	**Query:** Fields contained in user-defined type (UDT) **Purpose:** Returns the fields that are part of the structured UDTs in the input list **Syntax:** `stm_r_fd_contained_in_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`

Input List Type: fd

stm_r_fd_array_fd	**Query:** Fields by subtype **Purpose:** Returns the fields in the input list that are defined as array **Syntax:** `stm_r_fd_array_fd (IN el_list: LIST OF FIELD, OUT status: INTEGER)`
stm_r_fd_array_missing_fd	**Query:** Array of fields by subtype **Purpose:** Returns the array of fields in the input list for which no type is defined **Syntax:** `stm_r_fd_array_missing_fd (IN el_list: LIST OF FIELD, OUT status: INTEGER)`
stm_r_fd_bit_fd	**Query:** Fields by subtype **Purpose:** Returns the fields in the input list that are defined as bit **Syntax:** `stm_r_fd_bit_fd (IN el_list: LIST OF FIELD, OUT status: INTEGER)`
stm_r_fd_bit_queue_fd	**Query:** Fields by subtype **Purpose:** Returns the fields in the input list that are defined as queue of bit **Syntax:** `stm_r_fd_bit_queue_fd (IN el_list: LIST OF FIELD, OUT status: INTEGER)`
stm_r_fd_bits_array_fd	**Query:** Fields by subtype **Purpose:** Returns the fields in the input list that are defined as bit array **Syntax:** `stm_r_fd_bits_array_fd (IN el_list: LIST OF FIELD, OUT status: INTEGER)`

stm_r_fd_bits_fd	**Query:** Fields by subtype **Purpose:** Returns the fields in the input list that are defined as bit array **Syntax:** `stm_r_fd_bits_fd (IN el_list: LIST OF FIELD, OUT status: INTEGER)`
stm_r_fd_bits_queue_fd	**Query:** Fields by subtype **Purpose:** Returns the fields in the input list that are defined as queue of bit array **Syntax:** `stm_r_fd_bits_queue_fd (IN el_list: LIST OF FIELD, OUT status: INTEGER)`
stm_r_fd_by_attributes_fd	**Query:** Fields by attribute **Purpose:** Returns the fields in the input list that match the specified attribute and value `Syntax: stm_r_fd_by_attributes_fd (IN fd_list: LIST OF FIELD, IN attr_name: STRING, IN attr_value: STRING, OUT status: INTEGER)`
stm_r_fd_by_structure_type_fd	**Query:** None **Purpose:** Returns the fields in the input list that have the specified structure type (for example, single or array) **Syntax:** `stm_r_fd_by_structure_type_fd (IN el_list: LIST OF FIELD, IN dtype: INTEGER, OUT status: INTEGER)`
stm_r_fd_condition_fd	**Query:** Fields by subtype **Purpose:** Returns the fields in the input list that are defined as condition **Syntax:** `stm_r_fd_condition_fd (IN el_list: LIST OF FIELD, OUT status: INTEGER)`
stm_r_fd_condition_array_fd	**Query:** Fields by subtype **Purpose:** Returns the fields in the input list that are defined as array of condition **Syntax:** `stm_r_fd_condition_array_fd (IN el_list: LIST OF FIELD, OUT status: INTEGER)`
stm_r_fd_condition_queue_fd	**Query:** Fields by subtype **Purpose:** Returns the fields in the input list that are defined as queue of condition **Syntax:** `stm_r_fd_condition_queue_fd (IN el_list: LIST OF FIELD, OUT status: INTEGER)`

stm_r_fd_explicit_defined_fd	**Query:** Fields explicitly defined **Purpose:** Returns the fields in the input list that are explicitly defined **Syntax:** `stm_r_fd_explicit_defined_fd (IN el_list: LIST OF FIELD, OUT status: INTEGER)`
stm_r_fd_integer_array_fd	**Query:** Fields by subtype **Purpose:** Returns the fields in the input list that are defined as array of integer **Syntax:** `stm_r_fd_integer_array_fd (IN el_list: LIST OF FIELD, OUT status: INTEGER)`
stm_r_fd_integer_fd	**Query:** Fields by subtype **Purpose:** Returns the fields in the input list that are defined as integer **Syntax:** `stm_r_fd_integer_fd (IN el_list: LIST OF FIELD, OUT status: INTEGER)`
stm_r_fd_integer_queue_fd	**Query:** Fields by subtype **Purpose:** Returns the fields in the input list that are defined as queue of integer **Syntax:** `stm_r_fd_integer_queue_fd (IN el_list: LIST OF FIELD, OUT status: INTEGER)`
stm_r_fd_missing_fd	**Query:** Fields by subtype **Purpose:** Returns the fields in the input list for which no type is defined **Syntax:** `stm_r_fd_missing_fd (IN el_list: LIST OF FIELD, OUT status: INTEGER)`
stm_r_fd_name_of_fd	**Query:** Fields whose names match a given pattern **Purpose:** Returns all fields whose name matches the specified pattern **Syntax:** `stm_r_fd_name_of_fd (IN pattern: STRING, OUT status: INTEGER)`
stm_r_fd_queue_fd	**Query:** Fields by subtype **Purpose:** Returns the fields in the input list that are defined as queue **Syntax:** `stm_r_fd_queue_fd (IN el_list: LIST OF FIELD, OUT status: INTEGER)`

stm_r_fd_queue_missing_fd	**Query:** Queues of field by subtype **Purpose:** Returns the queues of fields in the input list for which no type is defined **Syntax:** `stm_r_fd_queue_missing_fd (IN el_list: LIST OF FIELD, OUT status: INTEGER)`
stm_r_fd_real_array_fd	**Query:** Fields by subtype **Purpose:** Returns the fields in the input list that are defined as array of real **Syntax:** `stm_r_fd_real_array_fd (IN el_list: LIST OF FIELD, OUT status: INTEGER)`
stm_r_fd_real_fd	**Query:** Fields by subtype **Purpose:** Returns the fields in the input list that are defined as real **Syntax:** `stm_r_fd_real_fd (IN el_list: LIST OF FIELD, OUT status: INTEGER)`
stm_r_fd_real_queue_fd	**Query:** Fields by subtype **Purpose:** Returns the fields in the input list that are defined as queue of real **Syntax:** `stm_r_fd_real_queue_fd (IN el_list: LIST OF FIELD, OUT status: INTEGER)`
stm_r_fd_single_fd	**Query:** Fields by subtype **Purpose:** Returns the fields in the input list that are defined as single **Syntax:** `stm_r_fd_single_fd (IN el_list: LIST OF FIELD, OUT status: INTEGER)`
stm_r_fd_string_array_fd	**Query:** Fields by subtype **Purpose:** Returns the fields in the input list that are defined as array of string **Syntax:** `stm_r_fd_string_array_fd (IN el_list: LIST OF FIELD, OUT status: INTEGER)`
stm_r_fd_string_fd	**Query:** Fields by subtype **Purpose:** Returns the fields in the input list that are defined as string **Syntax:** `stm_r_fd_string_fd (IN el_list: LIST OF FIELD, OUT status: INTEGER)`

stm_r_fd_string_queue_fd	**Query:** Fields by subtype **Purpose:** Returns the fields in the input list that are defined as queue of string **Syntax:** `stm_r_fd_string_queue_fd (IN el_list: LIST OF FIELD, OUT status: INTEGER)`
stm_r_fd_user_type_array_fd	**Query:** Fields by subtype **Purpose:** Returns the fields in the input list that are defined as array of user-defined type **Syntax:** `stm_r_fd_user_type_array_fd (IN el_list: LIST OF FIELD, OUT status: INTEGER)`
stm_r_fd_user_type_fd	**Query:** Fields by subtype **Purpose:** Returns the fields in the input list that are defined as user-defined type **Syntax:** `stm_r_fd_user_type_fd (IN el_list: LIST OF FIELD, OUT status: INTEGER)`
stm_r_fd_user_type_queue_fd	**Query:** Fields by subtype **Purpose:** Returns the fields in the input list that are defined as queue of user-defined type **Syntax:** `stm_r_fd_user_type_queue_fd (IN el_list: LIST OF FIELD, OUT status: INTEGER)`

Input List Type: mx

stm_r_fd_contained_in_mx	**Query:** Fields contained in a given element **Purpose:** Returns the fields that are part of the structured elements (data-items and user-defined types) in the input list **Syntax:** `stm_r_fd_contained_in_mx (IN el_list: LIST OF ELEMENT, OUT status: INTEGER)`

Functions (fn)

This section documents the queries that return a list of functions.

Input List Type: ch

stm_r_fn_name_of_fn	**Query:** Function names that match a given pattern **Purpose:** Returns all the functions whose names match the specified pattern **Syntax:** `stm_r_fn_name_of_fn (IN pattern: STRING, OUT status: INTEGER)`
stm_r_fn_unresolved_in_ch	**Query:** Functions unresolved in a given chart **Purpose:** Returns the functions that are unresolved in the charts of the input list `Syntax: stm_r_fn_unresolved_in_ch (IN ch_list: LIST OF CHART,OUT status: INTEGER)`

Information-Flows (if)

This section documents the queries that return a list of information-flows.

Input List Type: af

stm_r_if_basic_flowing_af	**Query:** Basic information-flows flowing through a given a-flow-line. **Purpose:** Returns information-flows that are not decomposed to other information items, and are flowing through a-flow-lines in the input list. **Syntax:** `stm_r_if_basic_flowing_af (IN af_list: LIST OF A_FLOW_LINE, OUT status: INTEGER)`
stm_r_if_flowing_through_af	**Query:** Information-flows flowing through a given a-flow-line. **Purpose:** Returns the information-flows flowing through a-flow-lines in the input list. **Note:** This function returns the highest information-flows, as opposed to `stm_r_if_basic_flowing_af`, which returns the lowest level. **Syntax:** `stm_r_if_flowing_through_af (IN af_list: LIST OF A_FLOW_LINE, OUT status: INTEGER)`
stm_r_if_labeling_af	**Query:** Information-flows labeling a given a-flow-line. **Purpose:** Returns the information-flows that label a-flow-lines in the input list. **Syntax:** `stm_r_if_labeling_af (IN af_list: LIST OF A_FLOW_LINE, OUT status: INTEGER)`

Input List Type: ch

stm_r_if_def_or_unres_in_ch	**Query:** Information-flows defined or unresolved in a given chart **Purpose:** Returns the information-flows that are explicitly defined or unresolved in the charts of the input list **Syntax:** `stm_r_if_def_or_unres_in_ ch (IN ch_list: LIST OF CHART,OUT status: INTEGER)`
stm_r_if_defined_in_ch	**Query:** Information-flows defined in a given chart **Purpose:** Returns the information-flows that are explicitly defined in the charts of the input list **Syntax:** `stm_r_if_defined_in_ch (IN ch_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_if_unresolved_in_ch	**Query:** Information-flows unresolved in a given chart **Purpose:** Returns the information-flows that are unresolved in the charts of the input list **Syntax:** `stm_r_if_unresolved_in_ch (IN ch_list: LIST OF CHART,OUT status: INTEGER)`

Input List Type: co

stm_r_if_containing_co	**Query:** Information-flows containing a given condition **Purpose:** Returns the information-flows containing conditions from the input list (conditions appearing in the **Consists of** field of the information-flow's form) **Syntax:** `stm_r_if_containing_co (IN co_list: LIST OF CONDITION, OUT status: INTEGER)`

Input List Type: di

stm_r_if_containing_di	**Query:** Information-flows containing a given data-item **Purpose:** Returns the information-flows containing data-items from the input list (data-items appearing in the **Consists of** field of the information-flow's form) **Syntax:** `stm_r_if_containing_di (IN di_list: LIST OF DATA_ITEM, OUT status: INTEGER)`

Input List Type: ev

stm_r_if_containing_ev	**Query:** Information-flows containing a given event **Purpose:** Returns the information-flows containing events from the input list (events appearing in the **Consists of** field of the information-flow's form) **Syntax:** `stm_r_if_containing_ev (IN ev_list: LIST OF EVENT, OUT status: INTEGER)`

Input List Type: if

stm_r_if_basic_if	**Query:** Basic information-flows **Purpose:** Returns the information-flows in the input list that are basic (those not defined using other information-flows) **Syntax:** `stm_r_if_basic_if (IN el_list: LIST OF INFORMATION_FLOW, OUT status: INTEGER)`
stm_r_if_by_attributes_if	**Query:** Information-flows by attributes **Purpose:** Returns the information-flows in the input list that match a particular attribute name and value **Syntax:** `stm_r_if_by_attributes_if (IN if_list: LIST OF INFORMATION_FLOW, IN attr_name: STRING, IN attr_value: STRING, OUT status: INTEGER)`
stm_r_if_contained_in_if	**Query:** Information-flows contained in a given information-flow **Purpose:** Returns the information-flows that are contained in information-flows from the input list (as defined in the **Consists of** field) **Syntax:** `stm_r_if_contained_in_if (IN if_list: LIST OF INFORMATION_FLOW, OUT status: INTEGER)`
stm_r_if_containing_if	**Query:** Information-flows containing a given information-flow **Purpose:** Returns the information-flows that contain information-flows from the input list (as defined in the **Consists of** field) **Syntax:** `stm_r_if_containing_if (IN if_list: LIST OF INFORMATION_FLOW, OUT status: INTEGER)`
stm_r_if_explicit_defined_if	**Query:** Information-flows explicitly defined **Purpose:** Returns the information-flows of the input list that were explicitly defined **Syntax:** `stm_r_if_explicit_defined_if (IN if_list: LIST OF INFORMATION_FLOW, OUT status: INTEGER)`
stm_r_if_name_of_if	**Query:** Information-flow names that match a given pattern **Purpose:** Returns all the information-flows whose names match the specified pattern `Syntax: stm_r_if_name_of_if (IN pattern: STRING, OUT status: INTEGER)`

stm_r_if_synonym_of_if	**Query:** Information-flow synonyms that match a given pattern **Purpose:** Returns all the information-flows whose synonyms match the specified pattern **Syntax:** `stm_r_if_synonym_of_if (IN pattern: STRING, OUT status: INTEGER)`
stm_r_if_unresolved_if	**Query:** Unresolved information-flows **Purpose:** Returns the unresolved information-flows in the input list **Syntax:** `stm_r_if_unresolved_if (IN if_list: LIST OF INFORMATION_FLOW, OUT status: INTEGER)`

Input List Type: mf

stm_r_if_basic_flowing_mf	**Query:** Basic information-flows flowing through a given m-flow-line. **Purpose:** Returns the information-flows that are not decomposed to other information items and are flowing through m-flow-lines in the input list. **Syntax:** `stm_r_if_basic_flowing_mf (IN mf_list: LIST OF M_FLOW_LINE, OUT status: INTEGER)`
stm_r_if_flowing_through_mf	**Query:** Information-flows flowing through a given m-flow-line. **Purpose:** Returns the information-flow flowing through m-flow-lines in the input list. **Note:** This function returns the highest information-flows as opposed to `stm_r_if_basic_flowing_mf`, which returns the lowest level. **Syntax:** `stm_r_if_flowing_through_mf (IN mf_list: LIST OF M_FLOW_LINE, OUT status: INTEGER)`
stm_r_if_labeling_mf	**Query:** Information-flows labeling a given m-flow-line. **Purpose:** Returns the information-flow labeling m-flow-lines in the input list. **Syntax:** `stm_r_if_labeling_mf (IN mf_list: LIST OF M_FLOW_LINE, OUT status: INTEGER)`

M-Flow-Lines (bf, bm, lmf, mf)

This section documents the queries that return a list of m-flow-lines. The types are as follows:

- `bf`—Basic m-flow-lines
- `lmf`—Local m-flow-lines
- `mf`—Global (compound) m-flow-lines

Output List Type: bf

Input List Type: co

stm_r_bf_within_flows_co	**Query:** A-flow-lines through which a given condition flows **Purpose:** Returns the a-flow-lines through which conditions from the input list actually flow **Syntax:** `stm_r_bf_within_flows_co (IN co_list: LIST OF CONDITION, OUT status: INTEGER)`
stm_r_bf_within_labels_co	**Query:** A-flow-lines labeled by a given condition **Purpose:** Returns the a-flow-lines labeled with conditions in the input list **Syntax:** `stm_r_bf_within_labels_co (IN co_list: LIST OF CONDITION, OUT status: INTEGER)`

Input List Type: di

stm_r_bf_within_flows_di	**Query:** A-flow-lines through which a given data-item flows **Purpose:** Returns the a-flow-lines through which data-items from the input list actually flow **Syntax:** `stm_r_bf_within_flows_di (IN di_list: LIST OF DATA_ITEM, OUT status: INTEGER)`
stm_r_bf_within_labels_di	**Query:** A-flow-lines labeled by a given data-item **Purpose:** Returns the a-flow-lines labeled with data-items in the input list **Syntax:** `stm_r_bf_within_labels_di (IN di_list: LIST OF DATA_ITEM, OUT status: INTEGER)`

Input List Type: ev

stm_r_bf_within_flows_ev	**Query:** A-flow-lines through which a given event flows **Purpose:** Returns the a-flow-lines through which events from the input list actually flow **Syntax:** `stm_r_bf_within_flows_ev (IN ev_list: LIST OF EVENT, OUT status: INTEGER)`
stm_r_bf_within_labels_ev	**Query:** A-flow-lines labeled by a given event **Purpose:** Returns the a-flow-lines labeled with events in the input list **Syntax:** `stm_r_bf_within_labels_ev (IN ev_list: LIST OF EVENT, OUT status: INTEGER)`

Input List Type: if

stm_r_bf_within_flows_if	**Query:** A-flow-lines through which a given information-flow flows **Purpose:** Returns the a-flow-lines through which information-flows from the input list actually flow **Syntax:** `stm_r_bf_within_flows_if (IN if_list: LIST OF INFORMATION_FLOW, OUT status: INTEGER)`
stm_r_bf_within_labels_if	**Query:** A-flow-lines labeled by a given information-flow **Purpose:** Returns the a-flow-lines labeled with information-flows in the input list **Syntax:** `stm_r_bf_within_labels_if (IN if_list: LIST OF INFORMATION_FLOW, OUT status: INTEGER)`

Input List Type: mx

stm_r_bf_from_source_mx	**Query:** A-flow-lines whose source is a given element **Purpose:** Returns basic a-flow-lines that originate at elements in the input list **Syntax:** `stm_r_bf_from_source_mx (IN bl_list: LIST OF ELEMENT, OUT status: INTEGER)`
stm_r_bf_to_target_mx	**Query:** A-flow-lines whose target is a given element **Purpose:** Returns the basic a-flow-lines whose target is an element from the input list **Syntax:** `stm_r_bf_to_target_mx (IN bl_list: LIST OF ELEMENT, OUT status: INTEGER)`
stm_r_bf_within_flows_mx	**Query:** A-flow-lines through which a given element flows **Purpose:** Returns the a-flow-lines through which elements from the input list actually flow **Syntax:** `stm_r_bf_within_flows_mx (IN if_list: LIST OF ELEMENT, OUT status: INTEGER)`
stm_r_bf_within_labels_mx	**Query:** A-flow-lines labeled by a given elements **Purpose:** Returns the a-flow-lines labeled with elements in the input list **Syntax:** `stm_r_bf_within_labels_mx (IN di_list: LIST OF ELEMENT, OUT status: INTEGER)`

Output List Type: bm

Input List Type: mf

stm_r_bm_contained_in_mf	**Query:** M-flow-lines labeled by given elements. **Purpose:** Returns the basic m-flow-lines contained in the m-flow-lines in the input list. **Syntax:** `STM_R_BM_CONTAINED_IN_MF(IN mf_list: LIST OF M_FLOW_LINE, OUT status: INTEGER):LIST OF M_FLOW_LINE`

stm_r_bm_enter_om	**Query:** M-flow-lines labeled by given elements.
	Purpose: Returns the basic m-flow-lines that enter the module-occurrences in the input list.
	Syntax:
	`STM_R_BM_ENTER_OM(IN om_list: LIST OF ELEMENT, OUT status:`
	`INTEGER):LIST OF M_FLOW_LINE`

Output List Type: lmf

Input List Type: bm

stm_r_lmf_containing_bm	**Query:** M-flow-lines labeled by a given elements **Purpose:** Returns the a-flow-lines labeled with elements in the input list **Syntax:** `STM_R_MF_CONTAINING_BM(IN bm_list: LIST OF M_FLOW_LINE, OUT status: INTEGER):LIST OF M_FLOW_LINE`

Input List Type: md

stm_r_lmf_from_source_md	**Query:** M-flow-lines whose source is a given module within chart **Purpose:** Returns the local compound m-flow-lines (those within charts) whose source is a module from the input list **Syntax:** `stm_r_lmf_from_source_md (IN md_list: LIST OF MODULE, OUT status: INTEGER)`
stm_r_lmf_input_to_md	**Query:** M-flow-lines input to a given module within the chart **Purpose:** Returns all the local compound m-flow-lines that originate outside and terminate at (or inside) modules in the input list **Syntax:** `stm_r_lmf_input_to_md (IN md_list: LIST OF MODULE, OUT status: INTEGER)`
stm_r_lmf_output_from_md	**Query:** M-flow-lines output from a given module within that chart **Purpose:** Returns all the local compound m-flow-lines that originate at (or inside) and terminate outside modules in the input list **Syntax:** `stm_r_lmf_output_from_md (IN md_list: LIST OF MODULE, OUT status: INTEGER)`
stm_r_lmf_to_target_md	**Query:** M-flow-lines whose target is a given module **Purpose:** Returns the local m-flow-lines whose target is a module from the input list **Syntax:** `stm_r_lmf_to_target_md (IN md_list: LIST OF MODULE, OUT status: INTEGER)`

Input List Type: mf

stm_r_lmf_contained_in_mf	**Query:** None **Purpose:** Returns the local m-flow-lines that contain the global m-flow-lines in the input list **Syntax:** `stm_r_lmf_contained_in_mf (IN af_list: LIST OF M_FLOW_LINE, OUT status: INTEGER)`

Output List Type: mf

Input List Type: co

stm_r_mf_within_flows_co	**Query:** M-flow-lines through which a given condition flows **Purpose:** Returns the m-flow-lines through which conditions from the input list actually flow **Syntax:** `stm_r_mf_within_flows_co (IN co_list: LIST OF CONDITION, OUT status: INTEGER)`
stm_r_mf_within_labels_co	**Query:** M-flow-lines labeled by a given condition **Purpose:** Returns the m-flow-lines that are labeled by the conditions in the input list **Syntax:** `stm_r_mf_within_labels_co (IN co_list: LIST OF CONDITION, OUT status: INTEGER)`

Input List Type: di

stm_r_mf_within_flows_di	**Query:** M-flow-lines through which a given data-item flows **Purpose:** Returns the m-flow-lines through which data-items from the input list actually flow **Syntax:** `stm_r_mf_within_flows_di (IN di_list: LIST OF DATA_ITEM, OUT status: INTEGER)`
stm_r_mf_within_labels_di	**Query:** M-flow-lines labeled by a given data-item **Purpose:** Returns the m-flow-lines that are labeled by the data-items in the input list **Syntax:** `stm_r_mf_within_labels_di (IN di_list: LIST OF DATA_ITEM, OUT status: INTEGER)`

Input List Type: ev

stm_r_mf_within_flows_ev	**Query:** M-flow-lines through which a given event flows **Purpose:** Returns the m-flow-lines through which events from the input list actually flow **Syntax:** `stm_r_mf_within_flows_ev (IN ev_list: LIST OF EVENT, OUT status: INTEGER)`
stm_r_mf_within_labels_ev	**Query:** M-flow-lines labeled by a given event **Purpose:** Returns the m-flow-lines that are labeled by the events in the input list **Syntax:** `stm_r_mf_within_labels_ev (IN ev_list: LIST OF EVENT, OUT status: INTEGER)`

Input List Type: if

stm_r_mf_within_flows_if	**Query:** M-flow-lines through which a given information-flow flows **Purpose:** Returns the m-flow-lines through which information-flows from the input list actually flow **Syntax:** `stm_r_mf_within_flows_if (IN if_list: LIST OF INFORMATION_FLOW, OUT status: INTEGER)`
stm_r_mf_within_labels_if	**Query:** M-flow-lines labeled with a given information-flow **Purpose:** Returns the m-flow-lines that are labeled with information-flows in the input list **Syntax:** `stm_r_mf_within_labels_if (IN if_list: LIST OF INFORMATION_FLOW, OUT status: INTEGER)`

Input List Type: lmf

stm_r_mf_containing_lmf	**Query:** None **Purpose:** Returns the global m-flow-lines (which might spread over several charts) that contain the local m-flow-lines (those within charts) in the input list **Syntax:** `stm_r_mf_containing_lmf (IN af_list: LIST OF M_FLOW_LINE, OUT status: INTEGER)`

Input List Type: md

stm_r_mf_from_source_md	**Query:** M-flow-lines whose source is a given module **Purpose:** Returns the global compound m-flow-lines (those that might spread over several charts) whose source is a module from the input list **Syntax:** `stm_r_mf_from_source_md (IN md_list: LIST OF MODULE, OUT status: INTEGER)`
stm_r_mf_input_to_md	**Query:** M-flow-lines input to a given module **Purpose:** Returns all the global compound m-flow-lines that originate outside and terminate at (or inside) modules in the input list **Syntax:** `stm_r_mf_input_to_md (IN md_list: LIST OF MODULE, OUT status: INTEGER)`
stm_r_mf_output_from_md	**Query:** M-flow-lines output from a given module **Purpose:** Returns all the global compound m-flow-lines that originate at (or inside) and terminate outside modules in the input list **Syntax:** `stm_r_mf_output_from_md (IN md_list: LIST OF MODULE, OUT status: INTEGER)`
stm_r_mf_to_target_md	**Query:** M-flow-lines whose target is a given module **Purpose:** Returns the global compound m-flow-lines whose target is a module from the input list **Syntax:** `stm_r_mf_to_target_md (IN md_list: LIST OF MODULE, OUT status: INTEGER)`
stm_r_lmf_to_target_md	**Query:** M-flow-lines whose target is a given module **Purpose:** Returns the local compound m-flow-lines whose target is a module from the input list **Syntax:** `stm_r_lmf_to_target_md (IN md_list: LIST OF MODULE, OUT status: INTEGER)`

Input List Type: mx

stm_r_mf_within_flows_mx	**Query:** M-flow-lines through which a given element flows **Purpose:** Returns the m-flow-lines through which elements from the input list actually flow **Syntax:** `stm_r_mf_within_flows_mx (IN mx_list: LIST OF ELEMENT,OUT status: INTEGER)`
stm_r_mf_within_labels_mx	**Query:** M-flow-lines that are labeled by a given information-flow **Purpose:** Returns the m-flow-lines that are labeled with elements in the input list **Syntax:** `stm_r_mf_within_labels_mx (IN mx_list: LIST OF ELEMENT,OUT status: INTEGER)`

Modules (md)

This section documents the queries that return a list of modules.

Input List Type: ac

stm_r_md_carrying_out_ac	**Query:** Modules carrying out a given activity. **Purpose:** Returns the modules carrying out activities in the input list. The modules appear in the **Implemented by Module** field of an activity's form. **Syntax:** `stm_r_md_carrying_out_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`

Input List Type: ch

stm_r_md_def_or_unres_in_ch	**Query:** Modules defined or unresolved in a given chart **Purpose:** Returns the modules that are explicitly defined or unresolved in the charts of the input list **Syntax:** `stm_r_md_def_or_unres_in_ch (IN ch_list: LIST OF CHART,OUT status: INTEGER)`
stm_r_md_defined_in_ch	**Query:** Modules defined in a given chart **Purpose:** Returns the modules that are explicitly defined in the charts of the input list **Syntax:** `stm_r_md_defined_in_ch (IN ch_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_md_described_by_ch	**Query:** Modules described by a given activity-chart **Purpose:** Returns the modules described by activity-charts in the input list **Syntax:** `stm_r_md_described_by_ch (IN ch_list: LIST OF CHART, OUT status: INTEGER)`

stm_r_md_instance_of_ch	**Query:** Modules instance of a given chart **Purpose:** Returns the instance modules defined by the charts in the input list **Syntax:** `stm_r_md_instance_of_ch (IN ch_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_md_root_in_ch	**Query:** Root modules of a given chart **Purpose:** Returns the internally defined modules (of type `diagram`) attached to the charts in the input list **Syntax:** `stm_r_md_root_in_ch (IN ch_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_md_top_level_in_ch	**Query:** Top-level modules of a given chart **Purpose:** Returns the top level modules (not contained in any box) of the charts in the input list **Syntax:** `stm_r_md_top_level_in_ch (IN ch_list: LIST OF CHART,OUT status: INTEGER)`
stm_r_md_unresolved_in_ch	**Query:** Modules unresolved in a given chart **Purpose:** Returns the modules that are unresolved in the charts of the input list **Syntax:** `stm_r_md_unresolved_in_ch (IN ch_list: LIST OF CHART,OUT status: INTEGER)`

Input List Type: ds

stm_r_md_contains_ds	**Query:** Modules in which a given data-store resides. **Purpose:** Returns the modules in which data-stores from the input list resides. The modules appear in the **Resides in Module** field of a data-store's form. **Syntax:** `stm_r_md_contains_ds (IN ds_list: LIST OF DATA_STORE, OUT status: INTEGER)`

Input List Type: md

stm_r_md_basic_md	**Query:** Basic modules **Purpose:** Returns the modules in the input list that are basic modules (those that have no descendants) **Syntax:** `stm_r_md_basic_md (IN md_list: LIST OF MODULE, OUT status: INTEGER)`
stm_r_md_bus_md	**Query:** Bus modules **Purpose:** Returns the modules in the input list that are bus modules **Syntax:** `stm_r_md_bus_md (IN md_list: LIST OF MODULE, OUT status: INTEGER)`
stm_r_md_by_attributes_md	**Query:** Modules by attributes **Purpose:** Returns the modules in the input list that match a particular attribute name and value **Syntax:** `stm_r_md_by_attributes_md (IN md_list: LIST OF MODULE, IN attr_name: STRING, IN attr_value: STRING, OUT status: INTEGER)`
stm_r_md_control_md	**Query:** Control modules **Purpose:** Returns the modules in the input list that are control modules **Syntax:** `stm_r_md_control_md (IN md_list: LIST OF MODULE, OUT status: INTEGER)`
stm_r_md_def_of_instance_md	**Query:** Definition modules of a given module **Purpose:** Returns the definition modules (top level modules in a definition chart) for instances in the input list **Syntax:** `stm_r_md_def_of_instance_md (IN md_list: LIST OF MODULE, OUT status: INTEGER)`
stm_r_md_defined_environment_md	**Query:** Environment modules **Purpose:** Returns the modules from the input list that were defined as environment modules **Syntax:** `stm_r_md_defined_environment_md (IN md_list: LIST OF MODULE,OUT status: INTEGER)`

stm_r_md_environment_md	**Query:** Environment modules **Purpose:** Returns the modules in the input list that are environment modules **Syntax:** `stm_r_md_environment_md (IN md_list: LIST OF MODULE,OUT status: INTEGER)`
stm_r_md_explicit_defined_md	**Query:** Modules explicitly defined **Purpose:** Returns the modules of the input list that were explicitly defined **Syntax:** `stm_r_md_explicit_defined_md (IN md_list: LIST OF MODULE, OUT status: INTEGER)`
stm_r_md_external_md	**Query:** External modules **Purpose:** Returns the modules in the input list that are external **Syntax:** `stm_r_md_external_md (IN md_list: LIST OF MODULE,OUT status: INTEGER)`
stm_r_md_generic_instance_md	**Query:** Generic instance modules **Purpose:** Returns the modules in the input list that are instances of generic charts **Syntax:** `stm_r_md_generic_instance_md (IN md_list: LIST OF MODULE, OUT status: INTEGER)`
stm_r_md_instance_md	**Query:** Instance modules **Purpose:** Returns the instance modules from the modules in the input list **Syntax:** `stm_r_md_instance_md (IN md_list: LIST OF MODULE, OUT status: INTEGER)`
stm_r_md_instance_of_def_m	**Query:** Instance modules of a given definition module **Purpose:** Returns the instance modules for definition modules (top-level modules in a definition chart) in the input list **Syntax:** `stm_r_md_instance_of_def_md (IN md_list: LIST OF MODULE, OUT status: INTEGER)`
stm_r_md_library_md	**Query:** Library modules **Purpose:** Returns the modules from the input list that are library modules **Syntax:** `stm_r_md_library_md (IN md_list: LIST OF MODULE, OUT status: INTEGER)`

stm_r_md_logical_desc_of_md	**Query:** Logical descendants of a given module **Purpose:** Returns the logical descendants of the modules in the input list, taking into account the translation of instances to their definition charts **Syntax:** `stm_r_md_logical_desc_of_md (IN md_list: LIST OF MODULE, OUT status: INTEGER)`
stm_r_md_logical_parent_of_md	**Query:** Logical parent modules of a given module **Purpose:** Returns the logical parent modules of the modules in the input list, taking into account the translation of instances to their definition charts **Syntax:** `stm_r_md_logical_parent_of_md (IN md_list: LIST OF MODULE, OUT status: INTEGER)`
stm_r_md_logical_sub_of_md	**Query:** Logical submodules of a given module **Purpose:** Returns the logical submodules of the modules in the input list, taking into account the translation of instances to their definition charts **Syntax:** `stm_r_md_logical_sub_of_md (IN md_list: LIST OF MODULE, OUT status: INTEGER)`
stm_r_md_name_of_md	**Query:** Modules whose names match a given pattern **Purpose:** Returns all the modules whose names match the specified pattern **Syntax:** `stm_r_md_name_of_md (IN pattern: STRING, OUT status: INTEGER)`
stm_r_md_offpage_instance_md	**Query:** Offpage instance modules **Purpose:** Returns the modules in the input list that are instances of offpage charts **Syntax:** `stm_r_md_offpage_instance_md (IN md_list: LIST OF MODULE, OUT status: INTEGER)`
stm_r_md_physical_desc_of_md	**Query:** Physical descendants of a given module **Purpose:** Returns the physical descendants (those within the same chart) for the modules in the input list **Syntax:** `stm_r_md_physical_desc_of_md (IN md_list: LIST OF MODULE, OUT status: INTEGER)`

stm_r_md_physical_parent_of_md	**Query:** Physical parent modules of a given module **Purpose:** Returns the physical parent modules (those within the same chart) for the modules in the input list **Syntax:** `stm_r_md_physical_parent_of_md (IN md_list: LIST OF MODULE, OUT status: INTEGER)`
stm_r_md_physical_sub_of_md	**Query:** Physical submodules of a given module **Purpose:** Returns the physical submodules (those within the same chart) for the modules in the input list **Syntax:** `stm_r_md_physical_sub_of_md (IN md_list: LIST OF MODULE, OUT status: INTEGER)`
stm_r_md_regular_md	**Query:** Regular modules **Purpose:** Returns the modules from the input list that are regular modules (not environment or storage) **Syntax:** `stm_r_md_regular_md (IN md_list: LIST OF MODULE, OUT status: INTEGER)`
stm_r_md_resolved_to_ext_md	**Query:** Modules resolved to a given external module **Purpose:** Returns the modules (internal, external, or environment) to which the external modules in the input list are resolved **Syntax:** `stm_r_md_resolved_to_ext_md (IN md_list: LIST OF MODULE,OUT status: INTEGER)`
stm_r_md_storage_md	**Query:** Storage modules **Purpose:** Returns the modules from the input list that are storage modules **Syntax:** `stm_r_md_storage_md (IN md_list: LIST OF MODULE, OUT status: INTEGER)`
stm_r_md_synonym_of_md	**Query:** Modules whose synonyms match a given pattern **Purpose:** Returns all the modules whose synonyms match the specified pattern **Syntax:** `stm_r_md_synonym_of_md (IN pattern: STRING, OUT status: INTEGER)`
stm_r_md_unresolved_md	**Query:** Unresolved modules **Purpose:** Returns the unresolved modules in the input list **Syntax:** `stm_r_md_unresolved_md (IN md_list: LIST OF MODULE, OUT status: INTEGER)`

Input List Type: mf

stm_r_md_source_of_mf	**Query:** Modules that are sources of a given m-flow-line **Purpose:** Returns the modules that are sources of m-flow-lines from the input list **Syntax:** `stm_r_md_source_of_mf (IN mf_list: LIST OF M_FLOW_LINE, OUT status: INTEGER)`
stm_r_md_target_of_mf	**Query:** Modules that are targets of a given m-flow-line **Purpose:** Returns the modules that are targets of m-flow-lines from the input list **Syntax:** `stm_r_md_target_of_mf (IN mf_list: LIST OF M_FLOW_LINE, OUT status: INTEGER)`

Input List Type: router

stm_r_md_contains_router	**Query:** Modules in which a given router resides. **Purpose:** Returns the modules in which routers from the input list resides. The modules appear in the **Resides in Module** field of a router's form. **Syntax:** `stm_r_md_contains_router (IN router_list: LIST OF ROUTER, OUT status: INTEGER)`

Mixed (mx)

This section documents the queries that return a list of elements.

Input List Type: af

stm_r_mx_flowing_through_af	**Query:** Elements flowing through a given a-flow-line **Purpose:** Returns the information elements (conditions, events, data-items, and basic information-flows) that actually flow through the a-flow-lines in the input list **Syntax:** `stm_r_mx_flowing_through_af (IN af_list: LIST OF A_FLOW_LINE, OUT status: INTEGER)`
stm_r_mx_labeling_af	**Query:** Elements labeling a given a-flow-line **Purpose:** Returns the elements that label a-flow-lines in the input list **Syntax:** `stm_r_mx_labeling_af (IN af_list: LIST OF A_FLOW_LINE, OUT status: INTEGER)`
stm_r_mx_source_of_af	**Query:** Elements that are sources of a given a-flow-line **Purpose:** Returns the elements (activities and data-stores) that are sources of a-flow-lines in the input list **Syntax:** `stm_r_mx_source_of_af (IN af_list: LIST OF A_FLOW_LINE,OUT status: INTEGER)`
stm_r_mx_target_of_af	**Query:** Elements that are targets of a given a-flow-line **Purpose:** Returns the elements (activities and data-stores) that are sources of a-flow-lines in the input list **Syntax:** `stm_r_mx_target_of_af (IN af_list: LIST OF A_FLOW_LINE,OUT status: INTEGER)`
stm_r_mx_source_of_ba	**Query:** Elements affected by a given activity. **Purpose:** Returns the elements that are sources of basic a-flow-lines in the input list. **Syntax:** `stm_r_mx_source_of_ ba (stm_list in_list, int *status);`
stm_r_mx_target_of_ba	**Query:** Elements affected by a given activity. **Purpose:** Returns the elements that are targets of basic a-flow-lines in the input list. **Syntax:** `stm_r_mx_target_of_ ba (stm_list in_list, int *status);`

stm_r_mx_target_of_ba	**Query:**Elements that are targets of a given flow line. **Purpose:** Returns the elements that are targets of basic a-flow-lines in the input list **Syntax:** `STM_R_MX_TARGET_OF_BA(IN ba_list: LIST OF A_FLOW_LINE,OUT status: INTEGER): LIST OF ELEMENT;`

Input List Type: ac

stm_r_mx_affected_by_ac	**Query:** Elements affected by a given activity. **Purpose:** Returns the elements (data-items, conditions, and events) affected (modified or generated) by activities, in mini-specs, and combinational assignments in the input list. **Syntax:** `stm_r_mx_affected_by_ac (IN ac_list: LIST OF ACTIVITY,OUT status: INTEGER)`
stm_r_mx_influence_ac	**Query:** Elements that are referenced by or influence a given activity. **Purpose:** Returns the elements used in all levels by the activities in the input list. This includes all logical descendant activities, a-flow-lines that enter or exit these activities, elements that appear in the various fields of these activities, and in the labels of the flow-lines and their components. **Syntax:** `stm_r_mx_influence_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_mx_influenced_by_ac	**Query:** Elements that see, or are influenced by, a given activity. **Purpose:** Returns the elements that directly or indirectly use the activities in the input list. This query identifies all the elements, in all levels, that see or affect the input activities. **Syntax:** `stm_r_mx_influenced_by_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_mx_refer_to_ac	**Query:** Elements that see a given activity. **Purpose:** Returns the elements that directly see activities in the input list. This query identifies where input activities are used. **Syntax:** `stm_r_mx_refer_to_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`

stm_r_mx_referenced_by_ac	**Query:** Elements that are referenced by a given activity. **Purpose:** Returns the elements that appear in the activities of the input list. This includes all physical descendant activities, a-flow-lines that enter or exit these activities, elements that appear in the various fields of these activities, and in the labels of the flow-lines. **Syntax:** `stm_r_mx_referenced_by_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_mx_resolved_to_ext_ac	**Query:** Elements resolved to a given external activity. **Purpose:** Returns the activities and modules (internal, external, or environment) to which the external activities in the input list are resolved. **Syntax:** `stm_r_mx_resolved_to_ext_ac (IN ac_list: LIST OF ACTIVITY,OUT status: INTEGER)`
stm_r_mx_used_by_ac	**Query:** Elements used by a given activity. **Purpose:** Returns the elements (data-items, conditions, and events) used (evaluated) by activities in mini-specs and combinational assignments in the input list. **Syntax:** `stm_r_mx_used_by_ac (IN ac_list: LIST OF ACTIVITY,OUT status: INTEGER)`

Input List Type: actor

stm_r_actor_explicit_defined_actor	**Query:** **Purpose:** Extracts a list of elements from the input list that are explictly defined elements of the requested type **Syntax:** `stm_r_actor_explicit_defined_actor (stm_list actor_list, int *status);`

Input List Type: an

stm_r_mx_in_definition_of_an	**Query:** Elements appearing in the definition of a given action. **Purpose:** Returns the elements that appear in the **Definition** field of actions (in the action's form) in the input list. This query identifies the elements that are used directly by the actions in the input list. **Syntax:** `stm_r_mx_in_definition_of_an (IN an_list: LIST OF ACTION, OUT status: INTEGER)`
stm_r_mx_influence_value_of_an	**Query:** Elements that influence the value of a given action. **Purpose:** Returns the elements that appear in the **Definition** field of actions in the input list, and those that appear in the definitions of these elements (for all levels). This query identifies the elements that are used directly or indirectly by the actions in the input list. **Syntax:** `stm_r_mx_influence_value_of_an (IN an_list: LIST OF ACTION, OUT status: INTEGER)`
stm_r_mx_influenced_by_an	**Query:** Elements that see, or are influenced by, a given action. **Purpose:** Returns the elements that directly or indirectly use the actions in the input list. This query identifies all the elements, in all levels, that use the input actions. **Syntax:** `stm_r_mx_influenced_by_an (IN an_list: LIST OF ACTION, OUT status: INTEGER)`
stm_r_mx_refer_to_an	**Query:** Elements that see a given action. **Purpose:** Returns the elements that directly use the actions in the input list. This query identifies where the input actions appear. **Syntax:** `stm_r_mx_refer_to_an (IN an_list: LIST OF ACTION, OUT status: INTEGER)`

Input List Type: bb

stm_r_bb_explicit_defined_bb	**Query:** **Purpose:** Extracts a list of elements from the input list that are explictly defined elements of the requested type **Syntax:** `stm_r_bb_explicit_defined_bb (stm_list bb_list, int *status);`

Input List Type: bt

stm_r_mx_source_of_bt	**Query:** Elements affected by a given activity. **Purpose:** Returns the elements that are sources of basic transitions in the input list **Syntax:** `stm_r_mx_source_of_bt(stm_list in_list, int *status);`
stm_r_mx_target_of_bt	**Query**: Transitions containing the given basic transitions **Purpose**: Returns the compound transitions that contain the basic transitions in the input list **Syntax**: `stm_r_tr_containing_bt (IN bt_list: LIST OF TRANSITION, OUT status: INTEGER)`
stm_r_tr_containing_bt	**Query**: Transitions containing the given basic transitions **Purpose**: Returns the compound transitions that contain the basic transitions in the input list **Syntax**: `stm_r_tr_containing_bt (IN bt_list: LIST OF TRANSITION, OUT status: INTEGER)`
stm_r_bt_exit_from_st	**Query:** Basic transitions whose source is the specified state **Purpose:** Returns the basic transitions whose source is a state appearing in the input list **Syntax:** `stm_r_bt_exit_from_st (IN st_list: LIST OF STATE, OUT status: INTEGER)`

stm_r_bt_exit_from_cn	**Query**: Basic transitions whose source is the specified connector. **Purpose**: Returns the basic transitions whose source is a connector appearing in the input list **Syntax:** `stm_r_bt_exit_from_cn (IN cn_list: LIST OF CONNECTOR, OUT status: INTEGER)`
stm_r_bt_enter_st	**Query**: Basic transitions whose target is the specified state **Purpose**: Returns the basic transitions whose target is a state appearing in the input list **Syntax**: `stm_r_bt_enter_st (IN st_list: LIST OF STATE, OUT status: INTEGER)`
stm_r_bt_enter_cn	**Query**: Basic transitions whose target is the specified connector **Purpose**: Returns the basic transitions whose target is a connector appearing in the input list **Syntax**: `stm_r_bt_enter_cn (IN st_list: LIST OF CONNECTOR, OUT status: INTEGER)`
stm_r_bt_defined_in_ch	**Query**: Basic transitions **Purpose**: Returns the basic transitions defined in the input list of charts **Syntax**: `stm_r_bt_defined_in_ch(IN ch_list: LIST OF CHART, OUT status: INTEGER);`
stm_r_af_containing_ba	**Query**: A-flow-lines **Purpose**: Returns the a-flow-lines that contain the basic a-flow-lines in the input list **Syntax**: `stm_r_af_containing_ba(IN ba_list: LIST OF A_FLOW_LINE, OUT status: INTEGER);`

Input List Type: bm

stm_r_mx_source_of_bm	**Query:** Elements affected by a given activity. **Purpose:** Returns the elements that are sources of basic m-flow-lines in the input list **Syntax:** `stm_r_mx_source_of_ bm (stm_list in_list, int *status);`
stm_r_mx_target_of_bm	**Query:** Elements affected by a given activity. **Purpose:** Returns the elements that are targets of basic m-flow-lines in the input list **Syntax:** `stm_r_mx_target_of_ bm (stm_list in_list, int *status);`
stm_r_bm_exit_from_md	**Query**: Basic m-flow-lines whose source is a given module **Purpose**: Returns the basic m-flow-lines whose source is a module from the input list **Syntax**: `stm_r_bm_exit_from_md (IN md_list: LIST OF MODULE, OUT status: INTEGER);`
stm_r_bm_exit_from_cn	**Query**: Basic m-flow-lines whose source is a given connector **Purpose**: Returns the basic m-flow-lines whose source is a connector from the input list **Syntax:** `stm_r_bm_exit_from_cn (IN cn_list: LIST OF CONNECTOR, OUT status: INTEGER);`
stm_r_bm_enter_md	**Query**: Basic m-flow-lines whose target is a given module **Purpose**: Returns the basic m-flow-lines whose target is a module from the input list **Syntax**: `stm_r_bm_enter_md (IN md_list: LIST OF MODULE, OUT status: INTEGER);`
stm_r_bm_enter_cn	**Query**: Basic m-flow-lines whose target is a given connector **Purpose**: Returns the basic m-flow-lines whose target is a connector from the input list **Syntax**: `stm_r_bm_enter_cn(IN md_list: LIST OF CONNECTOR, OUT status: INTEGER);`

stm_r_bm_exit_from_om	**Query**: Basic m-flow-lines whose source is a given module-occurrence **Purpose**: Returns the basic m-flow-lines whose source is a module-occurrence from the input list **Syntax**: `stm_r_bm_exit_from_om (IN md_list: LIST OF ELEMENT, OUT status: INTEGER);`

Input List Type: ch

stm_r_mx_constant_parameter_ch	**Query:** Constant parameters in a given chart **Purpose:** Returns the constant formal parameters of generic charts and components in the input list **Syntax:** `stm_r_mx_constant_parameter_ch (IN ch_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_mx_def_or_unres_in_ch	**Query:** Elements that are defined or unresolved in a given chart **Purpose:** Returns the elements that are explicitly defined or unresolved in the charts of the input list **Syntax:** `stm_r_mx_def_or_unres_in_ch (IN ch_list: LIST OF CHART,OUT status: INTEGER)`
stm_r_mx_defined_in_ch	**Query:** Elements that are defined in a given chart **Purpose:** Returns the elements that are explicitly defined in the charts of the input list **Syntax:** `stm_r_mx_defined_in_ch (IN ch_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_mx_in_parameter_ch	**Query:** In parameters in a given chart **Purpose:** Returns the formal in parameters of generic charts and components in the input list **Syntax:** `stm_r_mx_in_parameter_ch (IN ch_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_mx_influence_value_of_ch	**Query:** Elements referenced or influenced by a given chart **Purpose:** Returns the elements that are used directly or indirectly (referenced or affected) by the charts of the input list **Syntax:** `stm_r_mx_influence_value_of_ch (IN ch_list: LIST OF CHART,OUT status: INTEGER)`

stm_r_mx_inout_parameter_ch	**Query:** Inout parameters in a given chart **Purpose:** Returns the formal inout parameters of generic charts and components in the input list **Syntax:** `stm_r_mx_inout_parameter_ch (IN ch_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_mx_instance_of_ch	**Query:** Element instance of a given chart **Purpose:** Returns the element instances defined by the charts in the input list **Syntax:** `stm_r_mx_instance_of_ch (IN ch_list: LIST OF CHART,OUT status: INTEGER)`
stm_r_mx_out_parameter_ch	**Query:** Out parameters in a given chart **Purpose:** Returns the formal out parameters of generic charts and components in the input list **Syntax:** `stm_r_mx_out_parameter_ch (IN ch_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_mx_parameter_of_ch	**Query:** Parameters in a given chart **Purpose:** Returns the formal parameters of generic charts and components in the input list **Syntax:** `stm_r_mx_parameter_of_ch (IN ch_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_mx_referenced_by_ch	**Query:** Elements referenced by a given chart **Purpose:** Returns the elements that appear in the charts of the input list **Syntax:** `stm_r_mx_referenced_by_ch (IN ch_list: LIST OF CHART,OUT status: INTEGER)`
stm_r_mx_root_in_ch	**Query:** Root elements of a given chart **Purpose:** Returns the internally defined elements (of type diagram) attached to the charts in the input list **Syntax:** `stm_r_mx_root_in_ch (IN el_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_mx_text_def_unres_in_ch	**Query:** Textual elements defined or unresolved in a given chart **Purpose:** Returns the textual elements that are explicitly defined or unresolved in the charts of the input list **Syntax:** `stm_r_mx_text_def_unres_in_ch (IN ch_list: LIST OF CHART,OUT status: INTEGER)`

stm_r_mx_text_unresolved_in_ch	**Query:** Textual elements unresolved in a given chart **Purpose:** Returns the textual elements that are unresolved in the charts of the input list **Syntax:** `stm_r_mx_text_unresolved_in_ch (IN ch_list: LIST OF CHART,OUT status: INTEGER)`
stm_r_mx_textual_defined_in_ch	**Query:** Textual elements that are defined in a given chart **Purpose:** Returns the textual elements that are explicitly defined in the charts of the input list **Syntax:** `stm_r_mx_textual_defined_in_ch (IN ch_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_mx_unresolved_in_ch	**Query:** Elements unresolved in a given chart **Purpose:** Returns elements that are unresolved in the charts of the input list **Syntax:** `stm_r_mx_unresolved_in_ch (IN ch_list: LIST OF CHART,OUT status: INTEGER)`

Input List Type: co

stm_r_mx_in_definition_of_co	**Query:** Elements appearing in the definition of a given condition. **Purpose:** Returns the elements that appear in the **Definition** field of conditions (in the condition's form) in the input list. This query identifies the elements that are used directly by the conditions in the input list. **Syntax:** `stm_r_mx_in_definition_of_co (IN co_list: LIST OF CONDITION, OUT status: INTEGER)`
stm_r_mx_influence_value_of_co	**Query:** Elements that influence the value of a given condition. **Purpose:** Returns the elements that appear in the **Definition** field of the conditions in the input list, and those that appear in the definitions of these elements (for all levels). This query identifies the elements that directly or indirectly influence the conditions in the input list. **Syntax:** `stm_r_mx_influence_value_of_co (IN co_list: LIST OF CONDITION, OUT status: INTEGER)`

stm_r_mx_influenced_by_co	**Query:** Elements that see, or are influenced by, a given condition. **Purpose:** Returns the elements that directly or indirectly use the conditions in the input list. **Syntax:** `stm_r_mx_influenced_by_co (IN co_list: LIST OF CONDITION, OUT status: INTEGER)`
stm_r_mx_refer_to_co	**Query:** Elements that see a given condition. **Purpose:** Returns the elements that directly use the conditions in the input list. This query identifies where the input conditions appear. **Syntax:** `stm_r_mx_refer_to_co (IN co_list: LIST OF CONDITION, OUT status: INTEGER)`

Input List Type: di

stm_r_mx_in_definition_of_di	**Query:** Elements appearing in the definition of a given data-item. **Purpose:** Returns the elements that appear in the **Definition** and the **Consists of** fields of data-items (in the data-item's form) in the input list. This query identifies the elements that are directly used by the data-items in the input list. **Syntax:** `stm_r_mx_in_definition_of_di (IN di_list: LIST OF DATA_ITEM, OUT status: INTEGER)`
stm_r_mx_influence_value_of_di	**Query:** Elements that influence the value of a given data-item. **Purpose:** Returns the elements that appear in the **Definition** and **Consists** of fields of data-items in the input list, and those that appear in the fields of these elements (for all levels). This query identifies the elements that directly or indirectly influence the data-items in the input list. **Syntax:** `stm_r_mx_influence_value_of_di (IN di_list: LIST OF DATA_ITEM, OUT status: INTEGER)`
stm_r_mx_influenced_by_di	**Query:** Elements that see, or are influenced by, a given data-item. **Purpose:** Returns the elements that directly or indirectly use the data-items in the input list. This query identifies all the elements, in all levels, that see or affect the input data-items. **Syntax:** `stm_r_mx_influenced_by_di (IN di_list: LIST OF DATA_ITEM, OUT status: INTEGER)`

stm_r_mx_refer_to_di	**Query:** Elements that see the specified data-item. **Purpose:** Returns the elements that directly use the data-items in the input list. This query identifies where the input data-items appear. **Syntax:** `stm_r_mx_refer_to_di (IN di_list: LIST OF DATA_ITEM, OUT status: INTEGER)`

Input List Type: ds

stm_r_mx_refer_to_ds	**Query:** Elements that see a given data-store **Purpose:** Returns the elements directly affected by data-stores in the input list **Syntax:** `stm_r_mx_refer_to_ds (IN ds_list: LIST OF DATA_STORE, OUT status: INTEGER)`

Input List Type: dt

stm_r_mx_in_definition_of_dt	**Query:** Elements that appear in the definition of a given user-defined type **Purpose:** Returns the elements that appear in the definition form of the user-defined types in the input list **Syntax:** `stm_r_mx_in_definition_of_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`
stm_r_mx_influence_value_of_dt	**Query:** Elements that influence the definition of a given user-defined type **Purpose:** Returns the elements, data-items and user-defined types, that appear in the definition form of the user-defined types in the input list, and those that appear in the definition form of these elements—in all levels **Syntax:** `stm_r_mx_influence_value_of_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`

stm_r_mx_influenced_by_dt	**Query:** Elements that see or influenced by a given user-defined type **Purpose:** Returns the elements that directly or indirectly use in their definition the user-defined types in the input list. **Syntax:** `stm_r_mx_influenced_by_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`
stm_r_mx_refer_to_dt	**Query:** Elements that see a given user-defined type **Purpose**: Returns the elements that use in their definition form the user-defined types in the input list **Syntax:** `stm_r_mx_refer_to_dt (IN el_list: LIST OF DATA_TYPE, OUT status: INTEGER)`

Input List Type: ev

stm_r_mx_in_definition_of_ev	**Query:** Elements appearing in the definition of a given event. **Purpose:** Returns the elements that appear in the **Definition** field of events (in the event's form) in the input list. This query identifies the elements that are directly used by the events in the input list. **Syntax:** `stm_r_mx_in_definition_of_ev (IN ev_list: LIST OF EVENT, OUT status: INTEGER)`
stm_r_mx_influence_value_of_ev	**Query:** Elements that influence the value of a given event. **Purpose:** Returns the elements that appear in the **Definition** field of events in the input list, and those that appear in the definitions of these elements (for all levels). This query identifies the elements that are used directly or indirectly by the events in the input list. **Syntax:** `stm_r_mx_influence_value_of_ev (IN ev_list: LIST OF EVENT, OUT status: INTEGER)`
stm_r_mx_influenced_by_ev	**Query:** Elements that see, or are influenced by, a given event. **Purpose:** Returns the elements that directly or indirectly use the events in the input list. This query identifies all the elements, in all levels, that see or affect the input events. **Syntax:** `stm_r_mx_influenced_by_ev (IN ev_list: LIST OF EVENT, OUT status: INTEGER)`

stm_r_mx_refer_to_ev	**Query:** Elements that see a given event. **Purpose:** Returns the elements that directly use the events in the input list. This query identifies where the input events appear. **Syntax:** `stm_r_mx_refer_to_ev (IN ev_list: LIST OF EVENT, OUT status: INTEGER)`

Input List Type: fd

stm_r_mx_containing_fd	**Query:** Elements containing a given field. **Purpose:** Returns the data-items and user-defined types in which the fields in the input list are defined. **Syntax:** `stm_r_mx_containing_fd (IN el_list: LIST OF FIELD, OUT status: INTEGER)`
stm_r_mx_in_definition_of_fd	**Query:** Elements that appear in the definition of a given field. **Purpose:** Returns the elements that appear in the type definition of the fields in the input list. **Syntax:** `stm_r_mx_in_definition_of_fd (IN el_list: LIST OF FIELD, OUT status: INTEGER)`
stm_r_mx_influence_value_of_f	**Query:** Elements that influence the definition of a given field. **Purpose:** Returns the elements, data-items, and user-defined types that appear in the type definition of the fields in the input list, and those that appear in the definition form of these elements (in all levels). **Syntax:** `stm_r_mx_influence_value_of_fd (IN el_list: LIST OF FIELD, OUT status: INTEGER)`
stm_r_mx_influenced_by_fd	**Query:** Elements that see, or are influenced by, a given field. **Purpose:** Returns the elements that directly see the fields in the input list. This query identifies where the fields in the input list are used. **Syntax:** `stm_r_mx_influenced_by_fd (IN el_list: LIST OF FIELD, OUT status: INTEGER)`
stm_r_mx_refer_to_fd	**Query:** Elements that see a given field. **Purpose:** Returns the elements that directly see the fields in the input list. This query identifies where the fields in the input list are used. **Syntax:** `stm_r_mx_refer_to_fd (IN el_list: LIST OF FIELD, OUT status: INTEGER)`

Input List Type: fn

stm_r_mx_influenced_by_fn	**Query:** Elements that see, or are influenced by, a given function **Purpose:** Returns the elements that indirectly or directly use the functions in the input list. This query identifies all the elements, in all levels, that see the input functions. **Syntax:** `stm_r_mx_influenced_by_fn (IN fn_list: LIST OF FUNCTION, OUT status: INTEGER)`
stm_r_mx_refer_to_fn	**Query:** Elements that see a given function. **Purpose:** Returns the elements that directly use the functions in the input list. This query identifies where the input functions appear. **Syntax:** `stm_r_mx_refer_to_fn (IN fn_list: LIST OF FUNCTION, OUT status: INTEGER)`

Input List Type: if

stm_r_mx_in_definition_of_if	**Query:** Elements appearing in the definition of a given a information-flow. **Purpose:** Returns the elements listed in the **Consists of** field (in the information-flow's forms) for information-flows in the input list. This query identifies the elements that are used directly by the information-flows in the input list. **Syntax:** `stm_r_mx_in_definition_of_if (IN if_list: LIST OF INFORMATION_FLOW, OUT status: INTEGER)`
stm_r_mx_influence_value_of_if	**Query:** Elements that influence the value of a given information-flow. **Purpose:** Returns the elements contained in the information-flows in the input list (as listed in the **Consists of** field), for all levels of decomposition. This query identifies the elements that are directly or indirectly contained in the information-flows of the input list. **Syntax:** `stm_r_mx_influence_value_of_if (IN if_list: LIST OF INFORMATION_FLOW, OUT status: INTEGER)`

stm_r_mx_influenced_by_if	**Query:** Elements that see, or are influenced by, a given information-flow **Purpose:** Returns the elements that directly or indirectly use the information-flows in the input list. This query identifies all the elements, in all levels, that see the input information-flows. **Syntax:** `stm_r_mx_influenced_by_if (IN if_list: LIST OF INFORMATION_FLOW, OUT status: INTEGER)`
stm_r_mx_refer_to_if	**Query:** Elements that see a given information-flow. **Purpose:** Returns the elements that directly use the information-flows in the input list. This query identifies where the input information-flows appear. **Syntax:** `stm_r_mx_refer_to_if (IN if_list: LIST OF INFORMATION_FLOW, OUT status: INTEGER)`

Input List Type: md

stm_r_mx_influence_md	**Query:** Elements that are referenced by or influence a given module. **Purpose:** Returns the elements that are used in all levels by the modules in the input list. **Syntax:** `stm_r_mx_influence_md (IN md_list: LIST OF MODULE, OUT status: INTEGER)`
stm_r_mx_influenced_by_md	**Query:** Elements that see, or are influenced by, a given module **Purpose:** Returns the elements that directly or indirectly use the modules in the input list **Syntax:** `stm_r_mx_influenced_by_md (IN el_list: LIST OF MODULE, OUT status: INTEGER)`
stm_r_mx_refer_to_md	**Query:** Elements that see a given module. **Purpose:** Returns the elements that directly see modules in the input list. **Syntax:** `stm_r_mx_refer_to_md (IN md_list: LIST OF MODULE, OUT status: INTEGER)`

stm_r_mx_referenced_by_md	**Query:** Elements that are referenced by a given module. **Purpose:** Returns the elements that appear in the modules of the input list. **Syntax:** `stm_r_mx_referenced_by_md (IN md_list: LIST OF MODULE, OUT status: INTEGER)`
stm_r_mx_resolved_to_ext_md	**Query:** Elements resolved to a given external module. **Purpose:** Returns the elements to which the external modules in the input list are resolved. **Syntax:** `stm_r_mx_resolved_to_ext_md (IN md_list: LIST OF MODULE,OUT status: INTEGER)`

Input List Type: mf

stm_r_mx_flowing_through_mf	**Query:** Elements flowing through a given m-flow-line **Purpose:** Returns the information elements (conditions, events, data-items and basic information-flows) that actually flow through the m-flow-lines in the input list **Syntax:** `stm_r_mx_flowing_through_mf (IN mf_list: LIST OF M_FLOW_LINE, OUT status: INTEGER)`
stm_r_mx_labeling_mf	**Query:** Elements labeling a given m-flow-line **Purpose:** Returns the elements that label m-flow-lines in the input list **Syntax:** `stm_r_mx_labeling_mf (IN mf_list: LIST OF M_FLOW_LINE, OUT status: INTEGER)`

Input List Type: msg

stm_r_mx_labeling_msg	**Query:** Elements labeling a given message **Purpose:** Returns those elements that appear in labels of messages in the input list **Syntax:** `stm_r_mx_labeling_msg (IN mess_list: LIST OF MESSAGE, OUT status: INTEGER)`

Input List Type: mx

stm_r_mx_affected_by_mx	**Query:** Elements affected by a given element. **Purpose:** Returns the elements (primitive data-items, conditions, events, and activities) that are affected (modified, generated, started, stopped, and so on) by elements (states in static reactions or transitions in labels) in the input list. **Syntax:** `stm_r_mx_affected_by_mx (IN mx_list: LIST OF ELEMENT,OUT status: INTEGER)`
stm_r_mx_affecting_mx	**Query:** Elements in which a given element is affected. **Purpose:** Returns the elements (states and transitions) that affect (modify, generate, or activate) the elements (for example, events, data-items, or activities) in the input list. **Syntax:** `stm_r_mx_affecting_mx (IN mx_list: LIST OF ELEMENT,OUT status: INTEGER)`
stm_r_mx_meaningly_affecting_mx	**Query:** Activities in which a given element is affected. **Purpose:** Identical to stm_r_mx_affecting_mx, but when the input list includes an ID of a record/union, stm_r_mx_meaningly_affecting_mx will also return elements that affect a field of the record/union, and not necessarily the whole record/union element. **Syntax:** `stm_r_mx_meaningly_affecting_mx (stm_list in_list, int *status);`
stm_r_mx_by_attributes_mx	**Query:** Elements by attributes **Purpose:** Returns the elements in the input list that match a particular attribute name and value **Syntax:** `stm_r_mx_by_attributes_mx (IN mx_list: LIST OF ELEMENT, IN attr_name: STRING, IN attr_value: STRING, OUT status: INTEGER)`
stm_r_mx_callback_binding_mx	**Query:** Elements with callback bindings. **Purpose:** Returns the elements in the input list that have callback bindings. **Syntax:** `stm_r_mx_callback_binding_mx (IN el_list: LIST OF ELEMENT, OUT status: INTEGER)`

stm_r_mx_comb_elements_mx	**Query:** None. **Purpose:** Returns the elements (data-items and conditions) in the input list that are combinational elements. **Syntax:** `stm_r_mx_comb_elements_mx (IN mx_list: LIST OF ELEMENT, OUT status: INTEGER)`
stm_r_mx_def_of_instance_mx	**Query:** Definition elements of a given element. **Purpose:** Returns the definition elements (top-level) for instances in the input list. **Syntax:** `stm_r_mx_def_of_instance_mx (IN el_list: LIST OF ELEMENT, OUT status: INTEGER)`
stm_r_mx_explicit_defined_mx	**Query:** Elements explicitly defined. **Purpose:** Returns the elements of the input list that were explicitly defined. **Syntax:** `stm_r_mx_explicit_defined_mx (IN _list: LIST OF ELEMENT, OUT status: INTEGER)`
stm_r_mx_generic_instance_mx	**Query:** None. **Purpose:** Returns the boxes (states, activities, and modules) in the input list that are instances of generic charts. **Syntax:** `stm_r_mx_generic_instance_mx (IN mx_list: LIST OF ELEMENT,OUT status: INTEGER)`
stm_r_mx_in_definition_of_mx	**Query:** Elements that appear in the definition of a given element. **Purpose:** Returns the elements that appear in the various fields of the element's form, or in labels of elements in the input list. This query identifies the elements that are used directly by the elements in the input list. **Syntax:** `stm_r_mx_in_definition_of_mx (IN mx_list: LIST OF ELEMENT,OUT status: INTEGER)`
stm_r_mx_influence_value_of_mx	**Query:** Elements that influence the value of a given element. **Purpose:** Returns the elements that appear in various form's fields or labels of elements in the input list, and those that appear in the fields of these elements (for all levels). This query identifies the elements that directly or indirectly influence the elements in the input list. **Syntax:** `stm_r_mx_influence_value_of_mx` (IN mx_list: LIST OF ELEMENT,OUT status: INTEGER)

stm_r_mx_influenced_by_mx	**Query:** Elements that see or influenced by a given element. **Purpose:** Returns the elements that directly or indirectly use the elements in the input list. This query identifies all the elements, in all levels, that see or affect the input elements. **Syntax:** `stm_r_mx_influenced_by_mx (IN mx_list: LIST OF ELEMENT,OUT status: INTEGER)`
stm_r_mx_instance_mx	**Query:** Element instance of a given element **Purpose:** Returns the instance elements defined by the elements in the input list **Syntax:** `stm_r_mx_instance_mx (IN mx_list: LIST OF ELEMENT,OUT status: INTEGER)`
stm_r_mx_instance_of_def_mx	**Query:** Instance elements **Purpose:** Returns the instance elements for definition elements (top-level) in the input list. **Syntax:** `stm_r_mx_instance_of_def_mx (IN el_list: LIST OF ELEMENT, OUT status: INTEGER)`
stm_r_mx_logical_desc_of_mx	**Query:** Logical descendants of a given element **Purpose:** Returns the logical descendants of the elements in the input list **Syntax:** `stm_r_mx_logical_desc_of_mx (IN mx_list: LIST OF ELEMENT,OUT status: INTEGER)`
stm_r_mx_logical_parent_of_mx	**Query:** Logical parent elements of a given element. **Purpose:** Returns the logical parent elements of the elements in the input list, taking into account the translation of instances to their definition charts. **Syntax:** `stm_r_mx_logical_parent_of_mx (IN el_list: LIST OF ELEMENT, OUT status: INTEGER)`
stm_r_mx_logical_sub_of_mx	**Query:** Logical subelements of a given element **Purpose:** Returns the logical subelements of the elements in the input list **Syntax:** `stm_r_mx_logical_sub_of_mx (IN mx_list: LIST OF ELEMENT,OUT status: INTEGER)`

stm_r_mx_name_of_mx	**Query:** Element whose names match a given pattern **Purpose:** Returns all the elements whose names match the specified pattern **Syntax:** `stm_r_mx_name_of_mx (IN pattern: STRING, OUT status: INTEGER)`
stm_r_mx_offpage_instance_mx	**Query:** None. **Purpose:** Returns the boxes (states, activities, and modules) in the input list that are instances of offpage charts. **Syntax:** `stm_r_mx_offpage_instance_mx (IN mx_list: LIST OF ELEMENT,OUT status: INTEGER)`
stm_r_mx_parameter_mx	**Query:** Elements that are parameters. **Purpose:** Returns the elements in the input list that are declared as formal parameters of a generic chart. **Syntax:** `stm_r_mx_parameter_mx (IN mx_list: LIST OF ELEMENT,OUT status: INTEGER)`
stm_r_mx_physical_desc_of_mx	**Query:** Physical descendants of a given element **Purpose:** Returns the physical descendants (those within the same chart) for the elements in the input list **Syntax:** `stm_r_mx_physical_desc_of_mx (IN mx_list: LIST OF ELEMENT,OUT status: INTEGER)`
stm_r_mx_physical_parent_of_mx	**Query:** Physical parent elements of a given element **Purpose:** Returns the physical parent elements (those within the same chart) for the elements in the input list **Syntax:** `stm_r_mx_physical_parent_of_mx (IN mx_list: LIST OF ELEMENT,OUT status: INTEGER)`
stm_r_mx_physical_sub_of_mx	**Query:** Physical subelements of a given element **Purpose:** Returns the physical subelements (those within the same chart) for the elements in the input list **Syntax:** `stm_r_mx_physical_sub_of_mx (IN mx_list: LIST OF ELEMENT,OUT status: INTEGER)`
stm_r_mx_refer_to_mx	**Query:** Elements that see a given element. **Purpose:** Returns the elements that directly see elements in the input list. This query identifies where the input elements are used. **Syntax:** `stm_r_mx_refer_to_mx (IN mx_list: LIST OF ELEMENT,OUT status: INTEGER)`

stm_r_mx_resolved_to_ext_mx	**Query:** Elements resolved to a given external box. **Purpose:** Returns the activities and modules (internal, external, or environment) to which the external activities and modules in the input list are resolved. **Syntax:** `stm_r_mx_resolved_to_ext_mx (IN mx_list: LIST OF ELEMENT,OUT status: INTEGER)`
stm_r_mx_synonym_of_mx	**Query:** Elements whose synonyms match a given pattern **Purpose:** Returns all the elements whose synonyms match the specified pattern **Syntax:** `stm_r_mx_synonym_of_mx (IN pattern: STRING, OUT status: INTEGER)`
stm_r_mx_unresolved_mx	**Query:** Unresolved elements. **Purpose:** Returns the unresolved elements in the input list. **Syntax:** `stm_r_mx_unresolved_mx (IN mx_list: LIST OF ELEMENT, OUT status: INTEGER)`
stm_r_mx_used_by_mx	**Query:** Elements used by a given element. **Purpose:** Returns the elements (primitive events, conditions, data-items, states, and activities) that are used (evaluated by the elements, such as states in static reactions and transitions in labels) in the input list. **Syntax:** `stm_r_mx_used_by_mx (IN mx_list: LIST OF ELEMENT,OUT status: INTEGER)`
stm_r_mx_using_mx	**Query:** Elements in which a given element is used. **Purpose:** Returns the elements (states in static reactions and transitions in labels) that use (evaluate) the elements (basic events, conditions, data-items, states, and activities) in the input list. **Syntax:** `stm_r_mx_using_mx (IN mx_list: LIST OF ELEMENT,OUT status: INTEGER)`
stm_r_mx_meaningly_using_mx	**Query**: Activities in which a given element is used. **Purpose**: Identical to `stm_r_mx_using_mx`, but when the input list includes an ID of a record/union, `stm_r_mx_meaningly_using_mx` will also return elements that use a field of the record/union, and not necessarily the whole record/union element. **Syntax:** `stm_r_mx_meaningly_using_mx (stm_list in_list, int *status);`

stm_r_mx_with_combinationals_mx	**Query:** None. **Purpose:** Returns the elements (activities and state charts) in the input list that have combinational assignments. **Syntax:** `stm_r_mx_with_combinationals_mx (IN mx_list: LIST OF ELEMENT, OUT status: INTEGER)`

Input List Type: tr

stm_r_tr_by_attributes_tr	**Query**: Transitions by attributes **Purpose**: Returns the transitions from the input list which have attribute attr_name with value attr_value in their attribute list **Syntax**: `stm_r_tr_by_attributes_tr(IN st_list: LIST OF TRANSITION, IN attr_name : STRING, IN attr_value : STRING, OUT status: INTEGER);`

Input List Type: uc

stm_r_uc_explicit_defined_uc	**Query**: Use-case elements **Purpose**: Extracts the list of use-case elements from the input list that are explicitly defined **Syntax**: `stm_r_uc_explicit_defined_uc(IN uc_list: LIST OF USE_CASE, OUT status: INTEGER);`
stm_r_uc_associates_ac	**Query**: Use-case elements **Purpose**: Returns the use cases that associate with activities in the input list. **Syntax**: `stm_r_tr_by_attributes_tr(IN st_list: LIST OF TRANSITION, IN attr_name : STRING, IN attr_value : STRING, OUT status: INTEGER)`

Function Relationships

The following functions are related, but have subtle differences:

- `stm_r_mx_influenced_by_mx`
- `stm_r_mx_affected_by_mx`
- `stm_r_mx_used_by_mx`
- `stm_r_mx_affecting_mx`

The following matrix shows their relationships. In the matrix, opposite functions go from left to right, whereas cause and effect functions go up and down.

influenced by	used by
Function	
affected by	affecting

For example:

- If `x` is influenced by `y`, then `y` is used by `x`.
- If `n` is affected by `m`, then `m` is affecting `n`.

Consider the following statement:

```
if x is true then Function will set y=5
```

In this statement, `x` influences `Function` and `Function` affects `y`. This is shown in the matrix as follows:

- Elements above the double line influence `Function`.
- Elements below the double line are affected by `Function`.

For example, `x` is used by `Function` to determine whether to set the value of `y`, and `Function` is affecting `y` by setting its value.

There are four possible relationships between these functions: two opposites and two cause and effects.

- Opposite: influenced by and used by
- Opposite: affected by and affecting
- Cause and effect: influenced by and affected by
- Cause and effect: used by and affecting

To illustrate the relationships, consider the following static reaction in a state called `STATE`:

```
[D]/X=5 if D is true, then set x=y
```

The following table shows the relationships.

Relation Type	Description
Opposite: influenced and used by	STATE reads D to determine whether to perform an action, and D gives STATE the cue to set X=Y. In other words, STATE is influenced by D, and D is used by STATE.
Opposite: affected by and affecting	X's value is set by STATE and STATE sets the value of X. In other words, X is affected by STATE, and STATE is affecting X.
Cause and effect: influenced by and affected by	STATE reads y to determine which value should be assigned to X, and while in STATE, X can be set to y. In other words, when STATE is influenced by y, it results in X being affected by STATE.
Cause and effect: used by and affecting	If y is true, STATE sets the value of X. y is influencing STATE; STATE is affecting X. In other words, when y is used by STATE, it results in STATE affecting X.

Input List Type: router

stm_r_mx_flowing_from_router	**Query:** Elements flowing from a given router **Purpose:** Returns the elements actually flowing from routers in the input list **Syntax:** `stm_r_mx_flowing_from_router (IN router_lst: LIST OF ROUTER, OUT status: INTEGER)`
stm_r_mx_flowing_to_router	**Query:** Elements flowing to a given router **Purpose:** Returns the elements actually flowing to routers in the input list **Syntax:** `stm_r_mx_flowing_to_router (IN router_lst: LIST OF ROUTER, OUT status: INTEGER)`
stm_r_mx_refer_to_router	**Query:** Elements that see a given router. **Purpose:** Returns the elements that directly see routers in the input list. This query identifies where the routers are used. **Syntax:** `stm_r_mx_refer_to_router (IN router_list: LIST OF ROUTER, OUT status: INTEGER)`
stm_r_mx_resolved_to_ext_router	**Query:** Elements resolved to a given router. **Purpose:** Returns the elements to which the external routers in the input list are resolved. **Syntax:** `stm_r_mx_resolved_to_ext_router (IN router_list: LIST OF ROUTER, OUT status: INTEGER)`

Input List Type: sb

stm_r_mx_influenced_by_sb	**Query:** Elements that see, or are influenced by, a given subroutine **Purpose:** Returns the elements that directly or indirectly use the subroutines in the input list **Syntax:** `stm_r_mx_influenced_by_sb (IN fn_list: LIST OF SUBROUTINE, OUT status: INTEGER)`
stm_r_mx_refer_to_sb	**Query:** Elements that see a given subroutine **Purpose:** Returns the elements that directly see subroutines in the input list **Syntax:** `stm_r_mx_refer_to_sb (IN fn_list: LIST OF SUBROUTINE, OUT status: INTEGER)`

Input List Type: st

stm_r_mx_affected_by_st	**Query:** Elements affected by a given state. **Purpose:** Returns the elements (data-items, conditions, and events) that are affected (modified or generated) by states (in mini-specs and combinational assignments) in the input list. **Syntax:** `stm_r_mx_affected_by_st (IN st_list: LIST OF STATE,OUT status: INTEGER)`
stm_r_mx_influence_st	**Query:** Elements that are referenced by or influence a given state. **Purpose:** Returns the elements that are used in all levels by the states in the input list. This includes all logical descendant states, the transitions that enter or exit these states, and the elements that appear in the various fields of these states, the labels of the transitions, and their components. **Syntax:** `stm_r_mx_influence_st (IN st_list: LIST OF STATE, OUT status: INTEGER)`

stm_r_mx_influenced_by_st	**Query:** Elements that see, or are influenced by a given state. **Purpose:** Returns the elements that directly or indirectly use the states in the input list. This query identifies all the elements, in all levels, that see or affect the input states. **Syntax:** `stm_r_mx_influenced_by_st (IN st_list: LIST OF STATE, OUT status: INTEGER)`
stm_r_mx_refer_to_st	**Query:** Elements that see a given state. **Purpose:** Returns the elements that directly see states in the input list. This query identifies where the input states are used. **Syntax:** `stm_r_mx_refer_to_st (IN st_list: LIST OF STATE, OUT status: INTEGER)`
stm_r_mx_referenced_by_st	**Query:** Elements that are referenced by a given state. **Purpose:** Returns the elements that appear in the states of the input list. This includes all physical descendant states, the transitions that enter or exit these states, and the elements that appear in the various fields of these states and in the labels of the transitions. **Syntax:** `stm_r_mx_referenced_by_st (IN st_list: LIST OF STATE, OUT status: INTEGER)`
stm_r_mx_used_by_st	**Query:** Elements used by a given state. **Purpose:** Returns the elements (data-items, conditions, and events) that are used (evaluated) by states (in mini-specs and combinational assignments) in the input list. **Syntax:** `stm_r_mx_used_by_st (IN st_list: LIST OF STATE,OUT status: INTEGER)`

Input List Type: tr

stm_r_mx_affected_by_tr	**Query:** Elements affected by a given transition **Purpose:** Returns the elements (data-items, conditions, and events) that are affected (modified, generated) by transitions (in mini-specs and combinational assignments) in the input list **Syntax:** `stm_r_mx_affected_by_tr (IN tr_list: LIST OF TRANSITION,OUT status: INTEGER)`
stm_r_mx_labeling_tr	**Query:** Elements labeling a given transition **Purpose:** Returns those elements that appear in labels of the transitions in the input list **Syntax:** `stm_r_mx_labeling_tr (IN tr_list: LIST OF TRANSITION, OUT status: INTEGER)`
stm_r_mx_source_of_tr	**Query:** Elements that are sources of a given transition **Purpose:** Returns the elements (states and connectors) that are sources of transitions in the input list **Syntax:** `stm_r_mx_source_of_tr (IN tr_list: LIST OF TRANSITION,OUT status: INTEGER)`
stm_r_mx_target_of_tr	**Query:** Elements that are targets of a given transition **Purpose:** Returns elements (states and connectors) that are targets of transitions in the input list **Syntax:** `stm_r_mx_target_of_tr (IN tr_list: LIST OF TRANSITION,OUT status: INTEGER)`
stm_r_mx_used_by_tr	**Query:** Elements used by a given transition **Purpose:** Returns the elements (data-items, conditions, and events) that are used (evaluated) by transitions (in mini-specs and combinational assignments) in the input list **Syntax:** `stm_r_mx_used_by_tr (IN tr_list: LIST OF TRANSITION,OUT status: INTEGER)`

Input List Type: uc

stm_r_uc_explicit_defined_uc	**Query:** Elements from an input list. **Purpose:** Returns the elements (data-items, conditions, and events) that are explicitly defined. **Syntax:**
stm_r_uc_associates_uc	**Query:** **Purpose:** **Syntax:**

Routers (router)

This section documents the queries that return a list of routers.

Input List Type: ac

stm_r_router_contained_in_ac	**Query:** Routers contained in a given activity **Purpose:** Returns the routers contained directly in activities from the input list **Syntax:** `stm_r_router_contained_in_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_router_in_ac	**Query:** Router in a given activity **Purpose:** Returns the routers directly and indirectly contained in the activities from the input list **Syntax:** `stm_r_router_in_ac (IN el_list: LIST OF ROUTER, OUT status: INTEGER)`

Input List Type: af

stm_r_router_source_of_af	**Query:** Routers that are sources of a given a-flow-line **Purpose:** Returns the routers that are sources of a-flow-lines in the input list **Syntax:** `stm_r_router_source_of_af (IN af_list: LIST OF A_FLOW_LINE, OUT status: INTEGER)`
stm_r_router_target_of_af	**Query:** Routers that are targets of a given a-flow-line **Purpose:** Returns the routers that are sources of a-flow-lines in the input list **Syntax:** `stm_r_router_target_of_af (IN af_list: LIST OF A_FLOW_LINE, OUT status: INTEGER)`

Input List Type: ch

stm_r_router_def_or_unres_in_ch	**Query:** Routers defined or unresolved in a given chart **Purpose:** Returns the routers that are explicitly defined or unresolved in the charts of the input list **Syntax:** `stm_r_router_def_or_unres_in_ch (IN ch_list: LIST OF CHART,OUT status: INTEGER)`
stm_r_router_defined_in_ch	**Query:** Routers defined in a given chart **Purpose:** Returns the routers that are explicitly defined in the charts of the input list **Syntax:** `stm_r_router_defined_in_ch (IN ch_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_router_unresolved_in_ch	**Query:** Routers unresolved in a given chart **Purpose:** Returns the routers that are unresolved in the charts of the input list **Syntax:** `stm_r_router_unresolved_in_ch (IN ch_list: LIST OF CHART,OUT status: INTEGER)`

Input List Type: md

stm_r_router_resides_in_md	**Query:** Routers residing in a given module. **Purpose:** Returns the routers residing in modules from the input list. The module appears in the **Resides in Module** field of the router's form. **Syntax:** `stm_r_router_resides_in_md (IN md_list: LIST OF MODULE, OUT status: INTEGER)`

Input List Type: router

stm_r_router_by_attr_router	**Query:** Routers by attributes **Purpose:** Returns the routers in the input list that match a given attribute name and value **Syntax:** `stm_r_router_by_attr_router (IN router_list: LIST OF ROUTER, IN attr_name: STRING, IN attr_value: STRING, OUT status: INTEGER)`
stm_r_router_exp_def_router	**Query:** Routers that are explicitly defined **Purpose:** Returns from the input list those routers that were explicitly defined **Syntax:** `stm_r_router_exp_def_router (IN router_list: LIST OF ROUTER, OUT status: INTEGER)`
stm_r_router_name_of_router	**Query:** Routers whose names match a given pattern **Purpose:** Returns all routers whose name matches a given pattern **Syntax:** `stm_r_router_name_of_router (IN pattern: STRING, OUT status: INTEGER)`
stm_r_router_res_to_ext_router	**Query:** Routers resolved by a given external router **Purpose:** Returns the routers (internal and external) to which the external routers in the input list are resolved **Syntax:** `stm_r_router_res_to_ext_router (IN router_list: LIST OF ROUTER, OUT status: INTEGER)`

stm_r_router_synonym_of_router	**Query:** Routers whose synonyms match a given pattern **Purpose:** Returns all routers whose synonyms match a given pattern **Syntax:** `stm_r_router_synonym_of_router (IN pattern: STRING, OUT status: INTEGER)`
stm_r_router_unresolved_router	**Query:** Unresolved routers **Purpose:** Returns the unresolved routers in the input list **Syntax:** `stm_r_router_unresolved_router (IN router_list: LIST OF ROUTER, OUT status: INTEGER)`

Subroutines (sb)

This section documents the queries that return a list of subroutines.

Input List Type: ch

stm_r_sb_connected_to_ch	**Query:** Subroutines that are connected to a given procedural statechart **Purpose:** Returns the subroutines in the input list that are connected to the specified procedural statechart **Syntax:** `stm_r_sb_connected_to_ch (IN el_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_sb_defined_in_ch	**Query:** Subroutines defined in a given chart **Purpose:** Returns the subroutines that are explicitly defined in the charts in the input list **Syntax:** `stm_r_sb_defined_in_ch (IN el_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_sb_def_or_unres_in_ch	**Query:** Subroutines defined or unresolved in a given chart **Purpose:** Returns the subroutines that are explicitly defined or unresolved in the charts in the input list **Syntax:** `stm_r_sb_def_or_unres_in_ch (IN el_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_sb_unresolved_in_ch	**Query:** Subroutines unresolved in a given chart **Purpose:** Returns the subroutines that are unresolved in the charts in the input list **Syntax:** `stm_r_sb_unresolved_in_ch (IN ch_list: LIST OF CHART, OUT status: INTEGER)`

Input List Type: sb

stm_r_sb_ada_sb	**Query:** Subroutines written in Ada **Purpose:** Returns subroutines in the input list that are written in Ada and stored in the database using the Implementation menu **Syntax:** `stm_r_sb_ada_sb (IN el_list: LIST OF SUBROUTINE, OUT status: INTEGER)`
stm_r_sb_ansi_c_sb	**Query:** Subroutines written in ANSI C **Purpose:** Returns subroutines in the input list that are written in ANSI C and stored in the database using the Implementation menu **Syntax:** `stm_r_sb_ansi_c_sb (IN el_list: LIST OF SUBROUTINE, OUT status: INTEGER)`
stm_r_sb_bit_sb	**Query:** Subroutines by subtype **Purpose:** Returns the subroutines in the input list that are defined as bit **Syntax:** `stm_r_sb_bit_sb (IN el_list: LIST OF SUBROUTINE, OUT status: INTEGER)`
stm_r_sb_bits_sb	**Query:** Subroutines by subtype **Purpose:** Returns the subroutines in the input list that are defined as bit array **Syntax:** `stm_r_sb_bits_sb (IN el_list: LIST OF SUBROUTINE, OUT status: INTEGER)`
stm_r_sb_by_attributes_sb	**Query:** Subroutines by attributes **Purpose:** Returns the subroutines in the input list that match the specified attribute name and value `Syntax: stm_r_sb_by_attributes_sb (IN el_list: LIST OF SUBROUTINE, IN attr_name: STRING, IN attr_value: STRING, OUT status: INTEGER)`
stm_r_sb_explicit_defined_sb	**Query:** Subroutines that are explicitly defined **Purpose:** Returns the subroutines in the input list that are explicitly defined **Syntax:** `stm_r_sb_explicit_defined_sb (IN el_list: LIST OF SUBROUTINE, OUT status: INTEGER)`

stm_r_sb_fn_with_side_effect_sb	**Query:** Function subroutines with potential side-effects **Purpose:** Returns the function subroutines in the input list that have potential side-effects **Syntax:** `stm_r_sb_fn_with_side_effect_sb (IN el_list: LIST OF SUBROUTINE, OUT status: INTEGER)`
stm_r_sb_function_sb	**Query:** Subroutines defined as functions **Purpose:** Returns the subroutines in the input list that are defined as functions **Syntax:** `stm_r_sb_function_sb (IN el_list: LIST OF SUBROUTINE, OUT status: INTEGER)`
stm_r_sb_globals_usage_sb	**Query:** Subroutines that have global data **Purpose:** Returns all subroutines in the input list that have global data **Syntax:** `stm_r_sb_globals_usage_sb (IN el_list: LIST OF SUBROUTINE, OUT status: INTEGER)`
stm_r_sb_imp_action_lang_sb	**Query:** Subroutines whose selected implementation is Action Language **Purpose:** Returns the subroutines in the input list that are implemented in the Statemate Action Language using **Select Implementation** **Syntax:** `stm_r_sb_imp_action_lang_sb (IN el_list: LIST OF SUBROUTINE, OUT status: INTEGER)`
stm_r_sb_imp_ada_code_sb	**Query:** Subroutines whose selected implementation is Ada Code **Purpose:** Returns the subroutines in the input list that are implemented in Ada using **Select Implementation** in the properties **Syntax:** `stm_r_sb_imp_ada_code_sb (IN el_list: LIST OF SUBROUTINE, OUT status: INTEGER)`
stm_r_sb_imp_ansi_c_code_sb	**Query:** Subroutines whose selected implementation is ANSI C Code **Purpose:** Returns the subroutines in the input list that are implemented in ANSI C using **Select Implementation** **Syntax:** `stm_r_sb_imp_ansi_c_code_sb (IN el_list: LIST OF SUBROUTINE, OUT status: INTEGER)`

stm_r_sb_imp_best_match_sb	**Query:** Subroutines whose selected implementation is Best Match **Purpose:** Returns the subroutines in the input list that are implemented as the Best Match using **Select Implementation** **Syntax:** `stm_r_sb_imp_best_match_sb (IN el_list: LIST OF SUBROUTINE, OUT status: INTEGER)`
stm_r_sb_imp_kr_c_code_sb	**Query:** Subroutines whose selected implementation is K&R C Code **Purpose:** Returns the subroutines in the input list that are implemented in K&R C using **Select Implementation** **Syntax:** `stm_r_sb_imp_kr_c_code_sb (IN el_list: LIST OF SUBROUTINE, OUT status: INTEGER)`
stm_r_sb_imp_none_sb	**Query:** Subroutines whose selected implementation is None **Purpose:** Returns the subroutines in the input list that are not implemented (None) using **Select Implementation** **Syntax:** `stm_r_sb_imp_none_sb (IN el_list: LIST OF SUBROUTINE, OUT status: INTEGER)`
stm_r_sb_imp_procedural_sch_sb	**Query:** Subroutines whose selected implementation is Procedural Statechart **Purpose:** Returns the subroutines in the input list that are implemented as Procedural Statecharts using **Select Implementation** **Syntax:** `stm_r_sb_imp_procedural_sch_sb (IN el_list: LIST OF SUBROUTINE, OUT status: INTEGER)`
stm_r_sb_imp_truth_table_sb	**Query:** Subroutines implemented in a truth table **Purpose:** Returns the subroutines in the input list that were implemented in a truth table **Syntax:** `stm_r_sb_imp_truth_table_sb (IN el_list: LIST OF SUBROUTINE, OUT status: INTEGER)`
stm_r_sb_integer_sb	**Query:** Subroutines by subtype **Purpose:** Returns the subroutines in the input list that are defined as integer **Syntax:** `stm_r_sb_integer_sb (IN el_list: LIST OF SUBROUTINE, OUT status: INTEGER)`

stm_r_sb_kr_c_sb	**Query:** Subroutines written in K&R C **Purpose:** Returns subroutines in the input list that are written in K&R C and stored in the database using the Implementation menu **Syntax:** `stm_r_sb_kr_c_sb (IN el_list: LIST OF SUBROUTINE, OUT status: INTEGER)`
stm_r_sb_missing_sb	**Query:** Subroutines by subtype **Purpose:** Returns the subroutines in the input list for which no type is defined **Syntax:** `stm_r_sb_missing_sb (IN el_list: LIST OF SUBROUTINE, OUT status: INTEGER)`
stm_r_sb_name_of_sb	**Query:** Subroutines whose names match a given pattern **Purpose:** Returns all subroutines whose name matches the specified pattern **Syntax:** `stm_r_sb_name_of_sb (IN pattern: STRING, OUT status: INTEGER)`
stm_r_sb_parameters_sb	**Query:** Subroutines that have parameters **Purpose:** Returns all subroutines in the input list that have parameters **Syntax:** `stm_r_sb_parameters_sb (IN el_list: LIST OF SUBROUTINE, OUT status: INTEGER)`
stm_r_sb_procedural_sch_sb	**Query:** Subroutines designed as procedural statecharts **Purpose:** Returns subroutines in the input list that are designed as procedural statecharts and stored in the database using the Implementation menu **Syntax:** `stm_r_sb_procedural_sch_sb (IN el_list: LIST OF SUBROUTINE, OUT status: INTEGER)`
stm_r_sb_procedure_sb	**Query:** Subroutines defined as procedures **Purpose:** Returns the subroutines in the input list that are defined as procedures **Syntax:** `stm_r_sb_procedure_sb (IN el_list: LIST OF SUBROUTINE, OUT status: INTEGER)`
stm_r_sb_real_sb	**Query:** Subroutines by subtype **Purpose:** Returns the subroutines in the input list that are defined as real **Syntax:** `stm_r_sb_real_sb (IN el_list: LIST OF SUBROUTINE, OUT status: INTEGER)`

stm_r_sb_statemate_action_sb	**Query:** Subroutines written in the Statemate action language **Purpose:** Returns subroutines in the input list that are written in the Statemate action language and stored in the database using the Implementation menu **Syntax:** `stm_r_sb_statemate_action_sb (IN el_list: LIST OF SUBROUTINE, OUT status: INTEGER)`
stm_r_sb_string_sb	**Query:** Subroutines by subtype **Purpose:** Returns the subroutines in the input list that are defined as string **Syntax:** `stm_r_sb_string_sb (IN el_list: LIST OF SUBROUTINE, OUT status: INTEGER)`
stm_r_sb_synonym_of_sb	**Query:** Subroutines whose synonyms match a given pattern **Purpose:** Returns all subroutines whose synonyms match the specified pattern **Syntax:** `stm_r_sb_synonym_of_sb (IN el_list: LIST OF SUBROUTINE, OUT status: INTEGER)`
stm_r_sb_task_sb	**Query:** Subroutines defined as tasks **Purpose:** Returns the subroutines in the input list that are defined as tasks **Syntax:** `stm_r_sb_task_sb (IN el_list: LIST OF SUBROUTINE, OUT status: INTEGER)`
stm_r_sb_unresolved_sb	**Query:** Unresolved subroutines **Purpose:** Returns the unresolved subroutines in the input list **Syntax:** `stm_r_sb_unresolved_sb (IN el_list: LIST OF SUBROUTINE, OUT status: INTEGER)`
stm_r_sb_user_type_sb	**Query:** Subroutines by subtype **Purpose:** Returns the subroutines in the input list that are defined as user-defined type **Syntax:** `stm_r_sb_user_type_sb (IN el_list: LIST OF SUBROUTINE, OUT status: INTEGER)`
stm_r_sb_procedural_fch_sb	**Query**: Subroutines designed as Flowchart **Purpose**: Returns subroutines in the input list that are designed as Flowcharts and stored in the database using the Implementation menu **Syntax:** `stm_r_sb_procedural_fch_sb (IN sb_list: LIST OF SUBROUTINE, OUT st: INTEGER);`

stm_r_sch_connected_to_sb	**Query**: Statecharts connected to a given subroutine **Purpose**: Returns the procedural Statecharts that are connected to the subroutines in the input list **Syntax:** `stm_r_sch_connected_to_sb (IN sb_list: LIST OF SUBROUTINE, OUT st: INTEGER);`
stm_r_fch_connected_to_sb	**Query**: Flowcharts connected to a given subroutine **Purpose**: Returns the Flowcharts that are connected to the subroutines in the input list **Syntax:** `stm_r_fch_connected_to_sb (IN sb_list: LIST OF SUBROUTINE, OUT st: INTEGER);`

States (st)

This section documents the queries that return a list of states.

Input List Type: ac

stm_r_st_done_throughout_ac	**Query:** States in which a given activity is performed throughout **Purpose:** Returns the states for which activities in the input list are performed throughout that state (as specified in **Activities Within/ Throughout** field in the state's form) **Syntax:** `stm_r_st_done_throughout_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`
stm_r_st_done_within_ac	**Query:** States in which a given activity is performed within them **Purpose:** Returns the states in which activities in the input list are performed within that state (as specified in **Activities Within/ Throughout** field in the state's form) **Syntax:** `stm_r_st_done_within_ac (IN ac_list: LIST OF ACTIVITY, OUT status: INTEGER)`

Input List Type: ch

stm_r_st_def_or_unres_in_ch	**Query:** States defined or unresolved in a given chart **Purpose:** Returns the states that are explicitly defined or unresolved in the charts of the input list **Syntax:** `stm_r_st_def_or_unres_in_ch (IN ch_list: LIST OF CHART,OUT status: INTEGER)`
stm_r_st_defined_in_ch	**Query:** States defined in a given chart **Purpose:** Returns the states that are explicitly defined in the charts of the input list **Syntax:** `stm_r_st_defined_in_ch (IN ch_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_st_instance_of_ch	**Query:** State instances of a given chart **Purpose:** Returns the instance states defined by the charts in the input list **Syntax:** `stm_r_st_instance_of_ch (IN ch_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_st_root_in_ch	**Query:** Root states of a given chart **Purpose:** Returns the internally defined states (of type diagram) attached to the charts in the input list **Syntax:** `stm_r_st_root_in_ch (IN ch_list: LIST OF CHART, OUT status: INTEGER)`
stm_r_st_top_level_in_ch	**Query:** Top-level states of a given chart **Purpose:** Returns the top-level states (not contained in any box) of the charts in the input list **Syntax:** `stm_r_st_top_level_in_ch (IN ch_list: LIST OF CHART,OUT status: INTEGER)`
stm_r_st_unresolved_in_ch	**Query:** States unresolved in a given chart **Purpose:** Returns the states that are unresolved in the charts of the input list **Syntax:** `stm_r_st_unresolved_in_ch (IN ch_list: LIST OF CHART,OUT status: INTEGER)`

Input List Type: cn

stm_r_st_containing_cn	**Query:** States containing a given connector **Purpose:** Returns the states that encapsulate specified connectors from the input list **Syntax:** `stm_r_st_containing_cn (IN cn_list: LIST OF CONNECTOR,OUT status: INTEGER)`

Input List Type: mx

stm_r_st_affecting_mx	**Query:** States in which a given element is affected. **Purpose:** Returns the states that affect (modify, generate, or activate) the elements (for example, events, data-items, or activities) in the input list. **Syntax:** `stm_r_st_affecting_mx (IN mx_list: LIST OF ELEMENT,OUT status: INTEGER)`
stm_r_st_meaningly_affecting_mx	**Query**: States in which a given element is affected. **Purpose**: Identical to stm_r_st_affecting_mx, but when the input list includes an ID of a record/union, stm_r_st_meaningly_affecting_mx will also return elements that affect a field of the record/union, and not necessarily the whole record/union element. **Syntax**: `stm_r_st_meaningly_affecting_mx (IN mx_list: LIST OF ELEMENT,OUT status: INTEGER`
stm_r_st_using_mx	**Query:** States in which a given element is used. **Purpose:** Returns the states in static reactions that use (evaluate) the elements (basic events, conditions, data-items, states, and activities) in the input list. **Syntax:** `stm_r_st_using_mx (IN mx_list: LIST OF ELEMENT,OUT status: INTEGER)`
stm_r_st_meaningly_using_mx	**Query**: States in which a given element is used. **Purpose**: Identical to stm_r_st_using_mx, but when the input list includes an ID of a record/union, stm_r_st_meaningly_using_mx will also return elements that affect a field of the record/union, and not necessarily the whole record/union element. **Syntax**: `stm_r_st_meaningly_using_mx (IN mx_list: LIST OF ELEMENT,OUT status: INTEGER);`

Input List Type: st

stm_r_st_and_st	**Query:** And states. **Purpose:** Returns the states in the input list that are And-states. **Syntax:** `stm_r_st_and_st (IN st_list: LIST OF STATE, OUT status: INTEGER)`
stm_r_st_basic_st	**Query:** Basic states. **Purpose:** Returns the states in the input list that are basic (states that have no descendants). **Syntax:** `stm_r_st_basic_st (IN st_list: LIST OF STATE, OUT status: INTEGER)`
stm_r_st_by_attributes_st	**Query:** States by attributes. **Purpose:** Returns the states in the input list that match a given attribute name and value. **Syntax:** `stm_r_st_by_attributes_st (IN st_list: LIST OF STATE, IN attr_name: STRING, IN attr_value: STRING, OUT status: INTEGER)`
stm_r_st_callback_binding_st	**Query:** States with callback bindings. **Purpose:** Returns the states in the input list that have callback bindings. **Syntax:** `stm_r_st_callback_binding_st (IN el_list: LIST OF STATE, OUT status: INTEGER)`
stm_r_st_def_of_instance_st	**Query:** Definition states of a given state. **Purpose:** Returns the definition states (top-level in the definition chart) for instances in the input list. **Syntax:** `stm_r_st_def_of_instance_st (IN st_list: LIST OF STATE, OUT status: INTEGER)`
stm_r_st_default_entry_to_st	**Query:** Default entry to the default state. **Purpose:** Returns the default states of the or-states in the input list. **Syntax:** `stm_r_st_default_entry_to_st (IN st_list: LIST OF STATE, OUT status: INTEGER)`

stm_r_st_explicit_defined_st	**Query:** States explicitly defined. **Purpose:** Returns the states in the input list that were explicitly defined. **Syntax:** `stm_r_st_explicit_defined_st (IN st_list: LIST OF STATE, OUT status: INTEGER)`
stm_r_st_generic_instance_st	**Query:** Generic instance states. **Purpose:** Returns the states in the input list that are instances of generic charts. **Syntax:** `stm_r_st_generic_instance_st (IN st_list: LIST OF STATE, OUT status: INTEGER)`
stm_r_st_history_connector_st	**Query:** States containing a history connector. **Purpose:** Returns the states in the input list that contain a history connector. **Syntax:** `stm_r_st_history_connector_st (IN st_list: LIST OF STATE, OUT status: INTEGER)`
stm_r_st_instance_of_def_st	**Query:** Instance states of a given definition state. **Purpose:** Returns the instance states for definition states (top-level states in a definition chart) in the input list. **Syntax:** `stm_r_st_instance_of_def_st (IN st_list: LIST OF STATE, OUT status: INTEGER)`
stm_r_st_instance_st	**Query:** Instance states. **Purpose:** Returns those states in the input list that are instance states. **Syntax:** `stm_r_st_instance_st (IN st_list: LIST OF STATE, OUT status: INTEGER)`
stm_r_st_logical_desc_of_st	**Query:** Logical descendants of a given state. **Purpose:** Returns the logical descendants (children, grandchildren, and so on) of states in the input list, taking into account the translation of instances to their definition charts. **Syntax:** `stm_r_st_logical_desc_of_st (IN st_list: LIST OF STATE, OUT status: INTEGER)`

stm_r_st_logical_parent_of_st	**Query:** Logical parent states of a given state. **Purpose:** Returns the logical parent states of the states in the input list, taking into account the translation of instances to their definition charts. **Note:** This query provides similar output as `stm_r_st_physical_parent_of_st`, but for states that are substates of a top-level state in a definition chart, this query also returns the instance box. **Syntax:** `stm_r_st_logical_parent_of_st (IN st_list: LIST OF STATE, OUT status: INTEGER)`
stm_r_st_logical_sub_of_st	**Query:** Logical substates of a given state. **Purpose:** Returns the logical substates of the states in the input list, taking into account the translation of instances to their definition charts. **Syntax:** `stm_r_st_logical_sub_of_st (IN st_list: LIST OF STATE, OUT status: INTEGER)`
stm_r_st_name_of_st	**Query:** States whose names match a given pattern **Purpose:** Returns all the states whose names match the specified pattern **Syntax:** `stm_r_st_name_of_st (IN pattern: STRING, OUT status:INTEGER)`
stm_r_st_offpage_instance_st	**Query:** Offpage instance states. **Purpose:** Returns the states in the input list that are instances of offpage charts. **Syntax:** `stm_r_st_offpage_instance_st (IN st_list: LIST OF STATE, OUT status: INTEGER)`
stm_r_st_physical_desc_of_st	**Query:** Physical descendants of a given state. **Purpose:** Returns the physical descendants (those within the same chart) for the states in the input list. **Syntax:** `stm_r_st_physical_desc_of_st (IN st_list: LIST OF STATE, OUT status: INTEGER)`
stm_r_st_physical_parent_of_st	**Query:** Physical parent states of a given state. **Purpose:** Returns the physical parent states (those within the same chart) for the states in the input list. **Syntax:** `stm_r_st_physical_parent_of_st (IN st_list: LIST OF STATE, OUT status: INTEGER)`

stm_r_st_physical_sub_of_st	**Query:** Physical substates of a given state. **Purpose:** Returns the physical substates (those within the same chart) for the states in the input list. **Syntax:** `stm_r_st_physical_sub_of_st (IN st_list: LIST OF STATE, OUT status: INTEGER)`
stm_r_st_reaction_activity_st	**Query:** States having reactions or activities. **Purpose:** Returns the states from the input list that have static reactions or activities performed within or throughout the state. **Syntax:** `stm_r_st_reaction_activity_st (IN st_list: LIST OF STATE, OUT status: INTEGER)`
stm_r_st_synonym_of_st	**Query:** States whose synonyms match a given pattern **Purpose:** Returns all the states whose synonyms match the specified pattern **Syntax:** `stm_r_st_synonym_of_st (IN pattern: STRING, OUT status:INTEGER)`
stm_r_st_unresolved_st	**Query:** Unresolved states. **Purpose:** Returns the unresolved states in the input list. **Syntax:** `stm_r_st_unresolved_st (IN st_list: LIST OF STATE, OUT status: INTEGER)`

Input List Type: tr

stm_r_st_source_of_tr	**Query:** States that are sources of a given transition **Purpose:** Returns the states that are sources of transitions in the input list **Syntax:** `stm_r_st_source_of_tr (IN tr_list: LIST OF TRANSITION, OUT status: INTEGER)`
stm_r_st_target_of_tr	**Query:** States that are targets of a given transition **Purpose:** Returns the states that are targets of transitions in the input list **Syntax:** `stm_r_st_target_of_tr (IN tr_list: LIST OF TRANSITION, OUT status: INTEGER)`

Timing Constraint (tc)

This section documents the query that returns a list of timing constraints.

Input List Type: ch

stm_r_tc_defined_in_ch	**Query:** Timing constraints defined in a given chart **Purpose:** Returns the timing constraints that are explicitly defined in the charts of the input list **Syntax:** `stm_r_tc_defined_in_ch (IN ch_lst: LIST OF CHART, OUT status: INTEGER)`

Transitions (tr)

This section documents the queries that return a list of transitions.

Input List Type: cn

stm_r_tr_to_target_cn	**Query:** Transitions whose target is a given connector **Purpose:** Returns the transition in the input list whose target is a termination or history connector **Syntax:** `stm_r_tr_to_target_cn (IN cn_list: LIST OF CONNECTOR,OUT status: INTEGER)`

Input List Type: mx

stm_r_tr_affecting_mx	**Query:** Transitions in which a given element is affected. **Purpose:** Returns the transitions that affect (modify, generate, or activate) the elements (for example, events, data-items, or activities) in the input list. **Syntax:** `stm_r_tr_affecting_mx (IN mx_list: LIST OF ELEMENT,OUT status: INTEGER)`
stm_r_tr_meaningly_affecting_mx	**Query**: Transitions in which a given element is affected. **Purpose**: Identical to stm_r_tr_affecting_mx, but when the input list includes an ID of a record/union, stm_r_tr_meaningly_affecting_mx will also return elements that affect a field of the record/union, and not necessarily the whole record/union element. **Syntax**: `stm_r_tr_meaningly_affecting_mx (IN mx_list: LIST OF ELEMENT,OUT status: INTEGER);`
stm_r_tr_from_source_mx	**Query:** Transitions whose source is a given element **Purpose:** Returns transitions that originate at elements in the input list **Syntax:** `stm_r_tr_from_source_mx (IN mx_list: LIST OF ELEMENT,OUT status: INTEGER)`
stm_r_tr_to_target_mx	**Query:** Transitions whose target is a given element **Purpose:** Returns the transitions whose target is an element from the input list **Syntax:** `stm_r_tr_to_target_mx (IN mx_list: LIST OF ELEMENT,OUT status: INTEGER)`
stm_r_tr_using_mx	**Query:** Transitions in which a given element is used. **Purpose:** Returns the transitions in labels that use (evaluate) the elements (basic events, conditions, data-items, states, and activities) in the input list. **Syntax:** `stm_r_tr_using_mx (IN mx_list: LIST OF ELEMENT,OUT status: INTEGER)`

stm_r_tr_meaningly_using_mx	**Query**: Transitions in which a given element is used. **Purpose**: Identical to stm_r_tr_using_mx, but when the input list includes an ID of a record/union, stm_r_tr_meaningly_using_mx will also return elements that affect a field of the record/union, and not necessarily the whole record/union element. **Syntax**: `stm_r_tr_meaningly_using_mx (IN mx_list: LIST OF ELEMENT,OUT status: INTEGER);`

Input List Type: st

stm_r_tr_default_of_st	**Query:** Transitions that are the default entrance of a given state **Purpose:** Returns the default entrances (compound transitions) of the states in the input list **Syntax:** `stm_r_tr_default_of_st (IN st_list: LIST OF STATE,OUT status: INTEGER)`
stm_r_tr_from_source_st	**Query:** Transitions whose source is the specified state **Purpose:** Returns the transitions whose source is a state appearing in the input list **Syntax:** `stm_r_tr_from_source_st (IN st_list: LIST OF STATE, OUT status: INTEGER)`
stm_r_tr_to_target_st	**Query:** Transitions whose target is a given state **Purpose:** Returns the transitions whose target is a state appearing in the input list. **Syntax:** `stm_r_tr_to_target_st (IN st_list: LIST OF STATE, OUT status: INTEGER)`

Input List Type: tr

stm_r_tr_default_tr	**Query:** Default transition **Purpose:** Returns, of all the transitions in the input list that are default transitions **Syntax:** `stm_r_tr_default_tr (IN tr_list: LIST OF TRANSITION,OUT status: INTEGER)`

Utility Functions

Utility functions enable you to manipulate lists. For example, you could use utility functions to determine whether a particular element exists in a list of Statemate elements. Or, you could sort these lists to make reports easier to read. You can also use utility functions to manipulate strings of characters—to locate string patterns in a given string and to extract portions of strings. Most utility functions for lists can manipulate lists of any item type, but are usually used for lists of Statemate elements.

Utility functions do not extract information from the database; however, some utility functions use database information to complete their operations. These functions enable you to manipulate the information you have already retrieved using single-element or query functions.

In this appendix, the term *item* refers to any list member; the term element refers to a Statemate element.

The topics are as follows:

Calling Utility Functions

The calling sequence of the list functions is as follows:

```
stm_list_operation (list, status)
```

In this syntax:

- **stm_list**—Designates the function as a Statemate list manipulation function
- **operation**—The kind of list operation performed
- **list**—The list to be operated on
- **status**—The return function status code

The type of value returned by the function depends on the particular function. The returned value can be a list, a Statemate element, a string, or an integer. For example:

```
stm_list_sort_by_name (event_list, status)
```

This function alphabetically sorts the events in `event_list` according to their names. Function return values for each utility function are listed in **Single-Element Functions**.

The following sections document the utility functions that use a different calling sequence.

Contains Element

The `contains_element` function returns a Boolean value of `TRUE` if the item of interest is contained in the list (and `FALSE` if not contained).

The syntax is as follows:

```
stm_list_contains_element (list, item, status)
```

In this syntax:

- **stm_list**—Designates the function as a Statemate list manipulation function.
- **contains_element**—Signifies that this function checks to see whether the specified item is a member of the list.
- **list**—Is the list in which to search for the specified item. `list` can be any DGL list type (for example, `LIST OF ELEMENT`).
- **item**—Is an item whose presence in the list you want to verify. The data type of the item must be compatible with `list's` data type.
- **status**—Is the return function status code.

For example:

```
stm_list_contains_element (state_list, state_id, status)
```

This function returns a value of `TRUE` if the state whose ID is `state_id` is a member of the list `state_list`.

List Extraction by Type

The `list_extraction_by_type` function extracts the Statemate elements in the input list that are of a particular data type.

The syntax is as follows:

```
stm_list_extraction_by_type (element_type, list, status)
```

In this syntax:

- **stm_list**—Designates the function as a Statemate list manipulation function.
- **extraction_by_type**—Signifies that this function extracts all Statemate elements of a specified type from the input list.
- **element_type**—Is a Statemate element type predefined constant.
- **list**—Is the list in which to search for elements of the specified type. `list` can contain a variety of Statemate elements (a mixed list).
- **status**—Is the return function status code.

For example:

```
stm_list_extraction_by_type (stm_state, elmnt_list, status);
```

This call extracts all states from the list assigned to the variable `elmnt_list.` `stm_state` is one of the predefined constants that designates an element type.

List Extraction by Chart

The `list_extraction_by_chart` function extracts the Statemate elements in the input list that are of a particular chart.

The syntax is as follows:

```
stm_list_extraction_by_chart (chart_name, list, status)
```

In this syntax:

- **stm_list**—Designates the function as a Statemate list manipulation function.
- **extraction_by_chart**—Signifies that this function extracts all Statemate elements defined in a specific chart from the input list.
- **chart_name**—Is a string that represents the chart name.
- **list**—Is the list in which to search for elements of the specified chart. `list` can contain a variety of Statemate elements.
- **status**—Is the return function status code.

For example:

```
stm_list_extraction_by_chart ('my_chart', elmnt_list, status);
```

This call extracts all elements from the list that defined in (or unresolved in) the chart `my_chart.`

Location of Pattern in a String

The `index` function returns the first location from the left (starting with 0), in which `pattern` appears in the input string. It returns -1 when the pattern is not found.

The syntax is as follows:

```
stm_index (string, offset, pattern, status)
```

In this syntax:

- **stm_**—The standard prefix of Statemate functions.
- **index**—Signifies that the function looks for a location in the string.
- **string**—The input string in which the pattern is searched.
- **offset**—An integer greater than or equal to zero that represents the location at which the search begins.
- **pattern**—The string to search for.
- **status**—The return status code. The possible values are `stm_success` or `stm_null_string` (when uninitialized strings are used as parameters).

For example:

```
stm_index('ABCA',0,'C',status) = 2
stm_index('ABCA',0,'AB',status) = 0
stm_index('ABCA',2,'A',status) = 3
stm_index('ABCA',0,'AC',status) = -1
```

Extract Portion of a String

The `string_extract` function extracts a portion of a string.

The syntax is as follows:

```
stm_string_extract (string, index, length, status)
```

In this syntax:

- **stm_string**—Designates the function as a string manipulation function
- **extract**—Signifies that the function extracts a portion of the input string
- **string**—The input string
- **index**—The integer location from which the output string portion starts
- **length**—The output string length (integer)
- **status**—One of the following status codes: `stm_success`, `stm_null_string`, `stm_illegal_index`, and `stm_illegal_len`

For example:

```
stm_string_extract('ABCDE',0,2,status) = 'AB'
stm_string_extract('ABCDE',3,2,status) = 'DE'
stm_string_extract('ABCDE',3,7,status) = ''
```
and `status = stm_illegal_len`

Utility Function Input Arguments

The following table lists the input arguments used by the utility functions.

Argument	Description	DGL Data Type
list	List of items upon which you want to perform a logical or relational operation. The list's data type must be compatible with the particular utility function in which it appears as an argument.	`LIST OF...`
item	An item whose presence in the list you want to verify. The data type of the item must be compatible with list's data type.	Any
element type	A Statemate element type predefined constant (see **Rational Statemate Element Types**.)	Statemate element type
string	The string on which to perform a string manipulation function.	String
offset	The position (>= 0) in the string from which to start the specified manipulation.	Integer
pattern	The string portion searched for in the input string.	String
length	The length of the string portion extracted from the input string.	Integer

Examples of Utility Functions

This section shows how to use utility function calls to perform common tasks.

Utility Functions Example 1

The following example shows how to find the number of subactivities that exist for activity `A1`.

```
VARIABLE
   ACTIVITY          act_id, cntrl_act;
   LIST OF ACTIVITY  act_list, cntrl_act_list;
   INTEGER           status, list_length;
                     .
                     .
act_id := stm_r_ac ('A1', status);
act_list := stm_r_ac_physical_sub_of_ac ({act_id},
   status);
list_length := stm_list_length (act_list, status);
                     .
                     .
```

The example uses a single-element function to determine the ID of `A1`, then uses a query function to retrieve the list of `A1`'s subactivities. Finally, it assigns the number of `A1`'s subactivities to `list_length`.

Utility Functions Example 2

The following example is a continuation of the previous example. If you know that only one of the activities in the list `act_list` is a control activity and want to find out which activity it is, include the following code in your template:

```
          .
          .
cntrl_act_list := stm_r_ac_control_ac (act_list, status);
cntrl_act := stm_list_first_element (cntrl_act_list,
   status);
          .
          .
```

The list `cntrl_act_list` consists of only one element. This code extracts the first element (in this case, the only element) of the list and assigns this control activity's ID to `cntrl_act`.

Utility Functions Example 3

The following example shows how to find the software modules in a list of modules:

```
VARIABLE
   MODULE          md_id;
   LIST OF STRING  imp_type;
   INTEGER         status;
                    .
                    .
imp_type := stm_r_md_attr_val (md_id, 'IMPLEMENTATION',
   status);
IF stm_list_contains_string (imp_type, 'SOFTWARE',
   status)
   THEN
                    .
                    .
```

First, the code finds all the values for the `IMPLEMENTATION` attribute for the module, `md_id`. Among these values, the code searches for the value `SOFTWARE`. (Multiple values can exist for a given attribute.) If it is found, the statements following `THEN` are executed.

List of Utility Functions

In general, all the `stm_list` utilities work on lists of Statemate elements except strings. Strings have their own set of utilities, including:

```
stm_str_list_length (l_str, status)
stm_str_list_next_element (l_str, status)
stm_str_list_first_element (l_str, status)
stm_str_list_previous_element (l_str, status)
stm_str_list_last_element (l_str, status)
stm_list_contains_string (l_str, status)
```

The following pages document the utility functions. The functions are presented in alphabetical order, as listed in the following table.

stm_action_of_reaction	Extracts the action part of the specified reaction
stm_delete_file	Returns the definition type of the specified textual element
stm_dispose_memory	Frees temporary memory used by the DOC system.
stm_dispose_memory	Searches for the specified element in the list created by the search
stm_dispose_memory	Looks for the specified pattern in the given string
stm_int	Converts the specified real number to an integer
stm_int_to_string	Returns the string representation of the specified (decimal) number
stm_r_is_statemate	Returns TRUE if this is Statemate Classic
stm_list_contains_element	Determines whether the specified item appears in the given list

stm_list_contains_string	Determines whether the specified string appears in the given list
stm_list_extraction	Extracts the elements from the input list
stm_list_extraction_by_chart	Extracts the elements from the input list that belong to the specified chart
stm_list_extraction_by_type	Extracts elements of the specified type from the given list of Statemate elements
stm_list_first_element	Returns the first element in the specified list
stm_list_last_element	Returns the last item in the specified list
stm_list_length	Returns the length of the specified list
stm_list_next_element	Returns the next element in the specified list
stm_plot_ext	Generates a plot file with the indicated parameters,
stm_list_previous_element	Returns the previous element in the specified list
stm_list_sort	Alphabetically sorts the specified list of strings
stm_list_sort_by_attr_value	Sorts the specified list of Statemate elements by the value of the given attribute
stm_list_sort_by_branches	Sorts the list of hierarchical Statemate elements by branch
stm_list_sort_by_chart	Alphabetically sorts the input list of named Statemate elements by the name of the chart to which they belong
stm_list_sort_by_levels	Sorts the list of hierarchical Statemate elements by level
stm_list_sort_by_name	Sorts the list of Statemate elements alphabetically by name
stm_list_sort_by_synonym	Sorts the list of Statemate elements alphabetically by synonym
stm_list_sort_by_type	Sorts the list of Statemate elements by type
stm_multiline_to_one	Converts the specified multiline string (with new lines) to a one-line string (without new lines)
stm_multiline_to_strings	Converts the specified multiline expression to a list of strings
stm_plot	Generates a plot file with the indicated parameters, such as plot size, output device, and so on
stm_plot_hyper_exp	Generates the hyperlinks in a sequence diagram
stm_plot_with_autonumber	Prints a sequence diagram with numbered scenarios
stm_plot_with_break	Breaks a sequence diagram across multiple pages
stm_plot_with_headerline	Prints a sequence diagram with the names of lifelines on every page
stm_replace_string	Replaces the specified pattern with a new pattern within a string
stm_replace_word	Replaces the specified word with a new word within a given string
stm_set_rpt_formator	Returns the static reactions defined for the specified state element.
stm_str_list_first_element	Returns the first item in the specified list of strings
stm_str_list_last_element	Returns the element ID of the last item appearing in the specified list of strings
stm_str_list_length	Returns the number of items in the specified list of strings
stm_str_list_next_element	Returns the next item in the specified list of strings
stm_str_list_previous_element	Returns the previous item in the specified list of strings

stm_str_list_to_str	Returns the string representation of the specified list of strings
stm_str_to_list	Converts the specified string to a list of strings
stm_string_extract	Extracts a portion of the specified string
stm_string_free	Frees the memory used by the specified string
stm_string_to_int	Returns an integer value of a decimal string representation of a number
stm_string_retain	Frees temporary memory used by the DOC system.
stm_strlen	Returns the length of the specified string
stm_trigger_of_reaction	Returns the trigger part of a reaction

stm_action_of_reaction

Function type: `STRING`

Description

Extracts the action part of the specified reaction (the label of the transition or static reaction). The syntax of a reaction is *trigger/action*.

Note the following:

- The reaction is achieved by the single-element function `stm_r_st_reactions` or `stm_r_tr_labels`.
- The function returns an empty string when the action is missing.

Syntax

```
stm_action_of_reaction (reaction, status)
```

Arguments

Argument	Input/ Output	Type	Description
`reaction`	In	`STRING`	The reaction to decompose
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`

Example

Assume that `S1` has several static reactions and you want to list all the actions that are triggered when in this state. Include the following statements in your template:

```
VARIABLE
    STATE           st_id;
    INTEGER         status;
    LIST OF STRING  reactions;
    STRING          rct;
st_id := stm_r_st ('S1', status);
reactions := stm_r_st_reactions (st_id, status);
WRITE ('\n Actions of reactions in S1:');
FOR rct IN reactions LOOP
  WRITE ('\n', stm_action_of_reaction (rct, status));
END FOR;
```

stm_delete_file

Description:

Deletes the file.

Syntax

```
stm_delete_file (file_name_and_path, status)
```

Arguments

Argument	Input/Output	Type	Description
file_name_and_path	In	STRING	The file name and path.
`status`	Out	INTEGER	The function status code

Status Codes

- `stm_success`

stm_dispose_memory

Description:

When used in a DOC template, `STM_DISPOSE_MEMORY()` frees temporary memory used by the DOC system. If you want to avoid freeing a specific string, for example, general strings used in the template, call `STM_STRING_RETAIN(<str>)` on that string before calling `STM_DISPOSE_MEMORY()`.

Syntax

```
stm_dispose_memory()
```

Arguments

None

Status Codes

None

Example

Call the function with no parameters: `STM_DISPOSE_MEMORY();`

stm_index

Function type: `INTEGER`

Description

Looks for the specified pattern in the given string. It returns the first occurrence, starting from the offset position.

Note the following:

- The first position in the string is 0.
- If the pattern is not found in the string, the function returns -1.

Syntax

```
stm_index (string, offset, pattern, status)
```

Arguments

Argument	Input/Output	Type	Description
`string`	In	STRING	The string to be searched
`offset`	In	INTEGER	The starting position in the string
`pattern`	In	STRING	The pattern to look for
`status`	Out	INTEGER	The function status code

Status Codes

- `stm_success`
- `stm_null_string`

Example

```
stm_index('AB@CB',0,'B',status) = 1
stm_index('AB@CB',3,'B',status) = 4
stm_index('AB@CB',0,'XY',status) = -1
```

stm_int

Function type: `INTEGER`

Description

Converts the specified real number to an integer.

Syntax

```
stm_int(f)
```

Arguments

Argument	Input/Output	Type	Description
`f`	In	FLOAT	The floating-point number to convert

stm_int_to_string

Function type: `STRING`

Description

Returns the string representation of the specified (decimal) number.

Syntax

```
stm_int_to_string (int)
```

Arguments

Argument	Input/Output	Type	Description
`reaction`	In	INTEGER	The integer to be converted

Status Codes

- `stm_success`

Example

```
str := stm_int_to_string (36);
      str is '36'
```

stm_r_is_statemate

Function type: `BOOLEAN`

Description

Returns TRUE if this is Statemate Classic.

Syntax

```
stm_r_is_statemate()
```

stm_list_contains_element

Function type: `BOOLEAN`

Description

Determines whether the specified item appears in the given list.

Syntax

```
stm_list_contains_element (list, item, status)
```

Arguments

Argument	Input/ Output	Type	Description
`list`	In	LIST OF ELEMENT	The list to search.
`item`	In	ELEMENT	The item to look for. This can be a member of any element *except* strings. For strings, use `stm_list_contains_string`. Note that the item's data type must conform to the data type of the list. For example, if `list` is declared to be `LIST OF STATE`, `item` must be of type `STATE` (or `ELEMENT`). When the list consists of Statemate elements, `item` contains the element's ID.
`status`	Out	INTEGER	The function status code.

Status Codes

- stm_success
- stm_nil_list

Example

Suppose you want to check a list of Statemate elements for the presence of activity A1. The elements of interest are assigned to the list elmnt_list. Your template should contain the following statements:

```
VARIABLE
    ACTIVITY          act_id;
    LIST OF ELEMENT   elmnt_list;
    INTEGER           status;
                  .
                  .
                  .
act_id := stm_r_ac('A1', status);
                  .
                  .
                  .
IF stm_list_contains_element (elmnt_list, act_id, status) THEN
                  .
                  .
                  .
```

If A1 appears in elmnt_list, the statements following the IF statement are executed. Note that the ID of A1 is passed to the function, not its name).

stm_list_contains_string

Function type: `BOOLEAN`

Description

Determines whether the specified string appears in the given list.

Syntax

```
stm_list_contains_string (list, item, status)
```

Arguments

Argument	Input/Output	Type	Description
`list`	In	LIST OF ELEMENT	The list of strings to search
`item`	In	ELEMENT	The string to look for
`status`	Out	INTEGER	The function status code

Status Codes

- `stm_success`
- `stm_nil_list`

Example

Suppose you want to check a list of strings for the presence of a specific string. Your template should contain the following statements:

```
VARIABLE
    ACTIVITY          act_id;
    LIST OF STRING    string_list;
    INTEGER           status;
                    .
                    .
IF stm_list_contains_string
    (string_list, 'SPECIFICATION', status) THEN
                    .
                    .
```

If `'SPECIFICATION'` appears in `string_list`, the statements following the `IF` statement are executed.

stm_list_extraction

Function type: `LIST OF ELEMENT`

Description

Extracts the elements from the input list.

Syntax

```
stm_list_extraction (ex_type, el_list, status)
```

Arguments

Argument	Input/Output	Type	Description
`ex_type`	In	INTEGER	The type to extract from the list
`list`	In	LIST OF ELEMENT	The list of Statemate elements
`status`	Out	INTEGER	The function status code

Status Codes

- `stm_success`
- `stm_nil_list`

stm_list_extraction_by_chart

Function type: `LIST OF ELEMENT`

Description

Extracts the elements from the input list that belong to the specified chart.

Syntax

```
stm_list_extraction_by_chart (chart_name, list, status)
```

Arguments

Argument	Input/Output	Type	Description
`chart_name`	In	STRING	The name of the chart
`list`	In	LIST OF ELEMENT	The list of Statemate elements
`status`	Out	INTEGER	The function status code

Status Codes

- `stm_success`
- `stm_nil_list`
- `stm_illegal_name`
- `stm_name_not_found`

Example

To extract the list of all of Statemate elements that belong to the statechart `S8` from the input list `elem_list`, use the following statements:

```
VARIABLE
 LIST OF ELEMENT   elem_list, S8_elements;
 INTEGER           status;
 S8_elements:=
 stm_list_extraction_by_chart('S8', elem_list, status);
                   .
                    .
```

stm_list_extraction_by_chart_id

Function type: `LIST OF ELEMENT`

Description

Extracts the elements from the input list that belong to the specified chart.

Syntax

```
stm_list_extraction_by_chart (chart_name, list,status)
```

Arguments

Argument	Input/Output	Type	Description
`chart_name`	In	STRING	The name of the chart
`list`	In	LIST OF ELEMENT	The list of Statemate elements
`status`	Out	INTEGER	The function status code

Status Codes

- `stm_success`
- `stm_nil_list`
- `stm_illegal_name`
- `stm_name_not_found`

Example

To extract the list of all of Statemate elements that belong to the statechart `S8` from the input list `elem_list`, use the following statements:

```
VARIABLE
 LIST OF ELEMENT   elem_list, S8_elements;
 INTEGER           status;
 S8_elements:=
 stm_list_extraction_by_chart('S8', elem_list, status);
```

stm_list_extraction_by_type

Function type: `LIST OF ELEMENT`

Description

Extracts elements of the specified type from the given list of Statemate elements.

Syntax

```
stm_list_extraction_by_type (element_type, list, status)
```

Arguments

Argument	Input/ Output	Type	Description
`element_type`	In	STRING	The type to look for. `element_type` is one of the possible values of the enumerated type `stm_element_type`. The values of this type usually take the form `stm_element_type` (for example, `stm_state`, `stm_event`, and so on).
`list`	In	LIST OF ELEMENT	The list of elements (mixed types).
`status`	Out	INTEGER	The function status code.

Status Codes

- `stm_success`
- `stm_nil_list`
- `stm_illegal_extract_type`

Example

Suppose you want to extract a list of all the activities appearing in a list of Statemate elements. The input list is assigned to the variable `elmnt_list`. Your template should contain the following statements:

```
VARIABLE
    ACTIVITY          act;
    LIST OF ACTIVITY  act_list;
    LIST OF ELEMENT   elmnt_list;
    INTEGER           status;
                    .
                    .
                    .
act_list := stm_list_extraction_by_type (stm_activity,
   elmnt_list, status);
WRITE ('\n The activities in the list are:');
FOR act IN act_list LOOP
    WRITE ('\n', stm_r_ac_name (act, status));
END LOOP;
                    .
                    .
                     .
```

The names of all the activities in `elmnt_list` are written to your document.

stm_list_first_element

Function type: `ELEMENT`

Description

Returns the first element in the specified list. This function can be applied to a list of any DGL data types.

Syntax

```
stm_list_first_element (list, status)
```

Arguments

Argument	Input/Output	Type	Description
`list`	In	LIST OF ELEMENT	The list of items. Items in the input list can be of any element type except string. For strings, see the function `stm_str_list_first_element`.
`status`	Out	INTEGER	The function status code.

Status Codes

- `stm_success`
- `stm_nil_list`
- `stm_list_element_does_not_exist`

Example

Assume you have a list of activities assigned to the variable `act_list`. You know that only one of the activities in the list can be a control activity. To find out which activity this is, include the following statements in your template:

```
                    .
                    .
cntrl_act_list = stm_ac_control_ac (act_list, status);
cntrl_act = stm_list_first_element (cntrl_act_list,
    status);
```

These statements extract the first element (in this case, the only element) of the list and assign this control activity's ID to `cntrl_act`.

stm_list_last_element

Function type: ELEMENT

Description

Returns the last item in the specified list. This function can be applied to a list of any DGL data type.

Syntax

```
stm_list_last_element (list, status)
```

Arguments

Argument	Input/ Output	Type	Description
list	In	LIST OF ELEMENT	The list of items. Items in the input list can be of any element type except string. For strings, see the function stm_str_list_first_element.
status	Out	INTEGER	The function status code.

Status Codes

- stm_success
- stm_nil_list
- stm_list_element_does_not_exist

Example

Assume you have a list of states in the order S1, S2, S3 and S4. The list is assigned to the variable state_list. To find the last item in this list (S4), use the following statements:

```
VARIABLE
    STATE            state_id;
    LIST OF STATE    state_list;
    INTEGER          status;
                  .
                  .
                  .
state_id := stm_list_last_element (state_list, status);
WRITE ('\n The last state in the list is:  ',
  stm_r_st_name (state_id, status));
```

stm_list_length

Function type: `INTEGER`

Description

Returns the length of the specified list.

Syntax

```
stm_list_length (list, status)
```

Arguments

Argument	Input/Output	Type	Description
`list`	In	LIST OF ELEMENT	Items in the input list can ve of any element type except string. For strings, see to **stm_str_list_length**.
`status`	Out	INTEGER	The function status code.

Status Codes

- `stm_success`
- `stm_nil_list`

Example

Assume you extracted all the events from the database whose name begins with EV. Before writing the list to your document, you want to make sure that it will not span more than 30 lines of text (one page length). Your template should contain the following statements:

```
VARIABLE
    EVENT            ev;
    LIST OF EVENT    ev_list;
    INTEGER          ev_list_len, status;
CONSTANT
    INTEGER page_len :=  30;
                   .
                   .
                   .
ev_list:-stm_re_ev_name_of_ev ('EV*', status);
ev_list_len := stm_list_length (ev_list, status);
IF ev_list_len < page_len
WRITE ('\n List of Events: ');
FOR ev IN ev_list LOOP
    WRITE ('\n', stm_ev_name (ev, status));
END LOOP;
END IF
```

The list is written to the document if there are less than 30 events whose name begins with "EV".

stm_list_next_element

Function type: `ELEMENT`

Description

Returns the next item in the specified list. This function can be applied to a list of any DGL data type.

Note that “next” refers to the item physically located after the current item in the list. The “current” item is determined using the utility function `stm_list_first_element`.

Syntax

```
stm_list_next_element (list, status)
```

Arguments

Argument	Input/Output	Type	Description
`list`	In	LIST OF ELEMENT	The list of items. Items in the input list can be of any element type except string. For strings, see the function `stm_str_list_next_element`.
`status`	Out	INTEGER	The function status code.

Status Codes

- `stm_success`
- `stm_nil_list`
- `stm_list_element_does_not_exist`

Example

Assume you have a list of states in the order `S1`, `S2`, `S3` and `S4`. The list is assigned to the variable `state_list`. You locate the state `S1` by calling `stm_list_first_element`. `S1` becomes the “current’ item. To find the next element in the list, use the following statements:

```
VARIABLE
    LIST OF STATE         state_list;
    STATE                 state_id;
    INTEGER               status;
                      .
                      .
state_id := stm_list_first_element (state_list, status);
WRITE ('\n The first state in the list is:  ',
   stm_r_st_name (state_id, status));
state_id := stm_list_next_element (state_list, status);
WRITE ('\n The second state in the list is: ',
   stm_r_st_name (state_id, status));
                      .
                      .
```

This function is often used in loop statements.

stm_plot_ext

Function type:

Description

Generates a plot file with the indicated parameters, such as plot size, output device, and so on. The plot parameters are the same for all the different plot types (statecharts, activity charts, or module charts).

The output is designated for a particular device (one of the output devices defined in Statemate). The destination of the plot output is specified by one of the parameters. If its destination is not specified, the plot is included as part of the output segment file.

The function can generate the hyperlinks in the chart, print a sequence diagram with numbered scenarios, break a sequence diagram across multiple pages and print a sequence diagram with the names of lifelines on every page.

Syntax

```
stm_plot_ext (id, plot_file_name, width, height, device, data_position,
title_position, title, actual_h, pages_in_x, pages_in_y,  page_index_x,
page_index_y, headerline_y, options)
```

Arguments

Argument	Input/ Output	Type	Description
id	IN	Stamate element	The ID number of the Statemate chart to be plotted.
plot_file_name	IN	STRING	The name of the file destination to which the plot is written. The operating system pathname conventions are followed. You can specify a full path name to any directory for which you have write access.If you specify a simple file name, the plot is written to your workarea. If you do not specify a value (''), the plot is included as part of the output segment
width	IN	FLOAT	The maximum possible width of the plot (in inches).
height	IN	FLOAT	The maximum possibl elength of the plot (in inches). If you specify a plot size (width and height parameters) that is larger than the paper size defined for the specific printer, the plot simply uses the maximum allowable height and width defined for that printer.

Argument	Input/ Output	Type	Description
device	IN	STRING	Specifies the plotting device. This can be a supported formatting language if the plot is to be handled by a formatting processing system that has its own graphics language.To configure a new plotter or printer, select **Utilities > Output Devices** from the main Statemate window. Plots created using the Word format in the Output Device dialog are RTF files.
`data_position`	IN	STRING	The position of the date. This is a string that indicates where to place the plot date. The possible values are as follows: • **stm_plt_none** - The date is not included. • **stm_plt_top** - The date is placed at the top of the plot. • **stm_plt_bottom** - The date is placed at the bottom of the plot
`title_position`	IN	STRING	The title position. This is a string that indicates where to plac the plot title. The possible values are as follows: • **stm_plt_none** - The title is not included. • **stm_plt_top** - The title is placed at the top of the plot. • **stm_plt_bottom** - The title is placed at the bottom of the plot.
`title`	IN	STRING	Specifies the title to be printed with the plot
`actual_h`	OUT	FLOAT	Specifies the actual height (in inches) of the plotted output.
`pages_in_x`	OUT	INTEGER	Specifies how many pages the tool attempted to break the SD intoalong the x-axis. Note that if pages_in_x==0 and pages_in_y==0, the tool calculates a break pages scheme and assigns these variables so they can be read by the user after the call.
`pages_in_y`	OUT	INTEGER	Specifies how many pages the tool attempted to break the SD into along the y-axis.
`page_index_in_x`	IN	INTEGER	Plots the ith page in the x-axis.
`page_index_in_y`	IN	INTEGER	Plots the ith page in the y-axis.
`headerline_y`	IN	FLOAT	Defines the vertical coordinate on the page of the header line. This is usually 1.0.
`options`	IN		A list of strings of the form 'key=value'. See notes below for supported options

Status Codes

This function may return one of the two following status codes:

- `stm_success`
- `stm_can_not_open_file`
- `stm_id_out_of_range`
- `stm_not_enough_memory`
- `stm_id_not_found`
- `stm_empty_chart`
- `stm_unknown_plotter`
- `stm_plot_failure`
- `stm_unresolved`
- `stm_illegal_parameter`
- `stm_plot_illegal_option_key`
- `stm_plot_illegal_option_val`

Notes

The following are valid values for the 'options' argument:

- `stm_plot_option_hyperlink_ext_act_to_graphics`

  a. **For External-Activity**:

     When this option is 'no', the External_activity is hyperlinked to the 'Dictionary' description, if it exists, of the Activity it resolves to. When the 'Dictionary' description is empty, no link is created.

     When this option is 'yes', the External-Activity is hyperlinked to the chart in which the Activity it resolves to is in. If the resolved Activity is an Off-Page Activity, the link is to the off-page chart. If the resolved Activity is an Instance of generic, the link is to the generic chart. If the External-Activity resolves to a higher-level unresolved External-Activity, then the link is to the Chart where the Upper most instance of this External-Activity. If the External-Activity does not resolve to any Activity, no hyperlink is created.

  b. **For External-Router:**

     When this options is 'no', External_router is hyperlinked to the 'Dictionary' description, if it exists, of the Router it resolves to. When the 'Dictionary' description is empty, no link is created, When this option is 'yes', External-Router is hyperlinked to the chart that the Router it resolves to is in.

- `hyperlink_lifeline_to_graphics`

  When this option is 'no', Lifelines are hyperlinked to the 'Dictionary' description, if it exists, of the Activity they resolve to. When this option is 'yes', Lifelines are hyperlinked to the chart that the Activity they resolve to are in. If the resolved Activity is an Off-Page Activity, the link is to the off-page chart. If the resolved Activity is an Instance of generic, the link is to the generic chart. If the Lifeline resolves to an unresolved External- Activity, no link is created. If the Lifeline does not resolve to any Activity, no hyperlink is created.

stm_list_previous_element

Function type: `ELEMENT`

Description

Returns the previous element in the specified list. This function can be applied to a list of any DGL data type.

Note that “previous” refers to the item physically located before the current item in the list. The “current” item is determined using the utility function `stm_list_last_element`.

Syntax

```
stm_list_previous_element (list, status)
```

Arguments

Argument	Input/Output	Type	Description
`list`	In	LIST OF ELEMENT	The list of items. Items in the input list can be of any element type except string. For strings, see **stm_str_list_previous_element**.
`status`	Out	INTEGER	The function status code.

Status Codes

- `stm_success`
- `stm_nil_list`
- `stm_list_element_does_not_exist`

Example

Assume you have a list of states in the order `S1`, `S2`, `S3` and `S4`. The list is assigned to the variable `state_list`. You locate the state `S4` by calling `stm_list_last_element`. `S4` becomes the “current” item. To find the previous element in the list, use the following statements:

```
VARIABLE
    LIST OF STATE   state_list;
    STATE           state_id;
    INTEGER         status;
            .
            .
state_id := stm_list_last_element (state_list, status);
state_id := stm_list_previous_element (state_list,
    status);
WRITE (’\n State of interest is: ’,
    stm_r_st_name (state_id, status));
            .
            .
```

This function is often used in loop statements.

stm_list_sort

Function type: `LIST OF STRING`

Description

Alphabetically sorts the specified list of strings.

Syntax

stm_list_sort (list, status)

Arguments

Argument	Input/Output	Type	Description
`list`	In	LIST OF ELEMENT	The list of strings to be sorted
`status`	Out	INTEGER	The function status code

Status Codes

- `stm_success`
- `stm_nil_list`

Example

Assume you generated a list of attributes for the state `WAIT` using single-element functions. This list is assigned to the variable `attr_list`. To sort the list items and write them to your document, use the following statements:

```
VARIABLE
     STRING             item;
     LIST OF STRING     attr_list, ord_attr_list;
     INTEGER            status;
                  .
                  .
WRITE ('\n WAIT's attributes are:');
ord_attr_list := stm_list_sort (attr_list, status);
FOR item IN ord_attr_list LOOP
    WRITE ('\n ', item);
END LOOP;
```

stm_list_sort_by_attr_value

Function type: `LIST OF ELEMENT`

Description

Sorts the specified list of Statemate elements by the value of the given attribute.

Note that the function receives and returns a list of element IDs, not a list of element names.

Syntax

```
stm_list_sort_by_attr_value (list, attr_name, status)
```

Arguments

Argument	Input/Output	Type	Description
`list`	In	LIST OF ELEMENT	The list of Statemate element IDs to be sorted.
`attr_name`	In	STRING	The attribute to use as the sorting key.
`status`	Out	INTEGER	The function status code. The function returns the status code `stm_elements_without_attributes` if you apply this function to a list of elements that do not have the specified attribute.

Status Codes

- `stm_success`
- `stm_nil_list`
- `stm_elements_without_attributes`

Example

Suppose you want to write a particular list of activities from the database to your document. You extract the activities of interest using single-element and query functions and build a list of such activities. This list is assigned to the variable `act_list`. To sort the activities by the value of an attribute called “code”, your template should contain the following function calls:

```
VARIABLE
   LIST OF ACTIVITY    act_list, ord_act_list;
   ACTIVITY            activ;
   INTEGER             status;
                .
                .
                .
ord_act_list := stm_list_sort_by_attr_value (act_list,
   'code', status);
WRITE ('\n Ordered list of activities:');
FOR activ IN ord_act_list LOOP
  WRITE ('\n', stm_r_ac_name (activ, status));
END LOOP;
```

stm_list_sort_by_branches

Function type: `LIST OF ELEMENT`

Description

Sorts the specified list of hierarchical Statemate elements by branches.

Note the following:

- This function is relevant only for a list of hierarchical elements. If the function is applied to a list of non-hierarchical elements, `status` receives the value `stm_elements_not_hierarchical`.
- The order in which branches appear in the output is arbitrary. However, the order of states appearing within each branch are ordered from top to bottom.

Syntax

```
stm_list_sort_by_branches (list, status)
```

Arguments

Argument	Input/Output	Type	Description
`list`	In	LIST OF ELEMENT	The list of Statemate elements
`status`	Out	INTEGER	The function status code

Status Codes

- `stm_success`
- `stm_nil_list`
- `stm_elements_not_hierarchical`

stm_list_sort_by_chart

Function type: `LIST OF ELEMENT`

Description

Alphabetically sorts the input list of named Statemate elements, by the name of the chart to which they belong. The input list consists of Statemate elements.

Note that this function receives and returns a list of element IDs, not a list of element names.

Syntax

```
stm_list_sort_by_chart (el_list, status)
```

Arguments

Argument	Input/Output	Type	Description
`el_list`	In	LIST OF ELEMENT	The list of Statemate box elements
`status`	Out	INTEGER	The function status code

Status Codes

- `stm_success`
- `stm_nil_list`

stm_list_sort_by_levels

Function type: `LIST OF ELEMENT`

Description

Sorts a list of hierarchical Statemate elements by level.

Note that this function is relevant only for a list of hierarchical elements. If the function is applied to a list of non-hierarchical elements, `status` receives the value `stm_elements_not_hierarchical`.

Syntax

```
stm_list_sort_by_levels (list, status)
```

Arguments

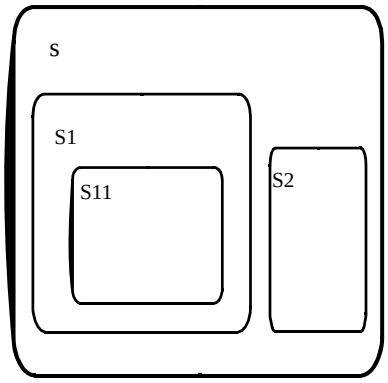
Argument	Input/Output	Type	Description
`list`	In	LIST OF ELEMENT	The list of Statemate box elements
`status`	Out	INTEGER	The function status code

Status Codes

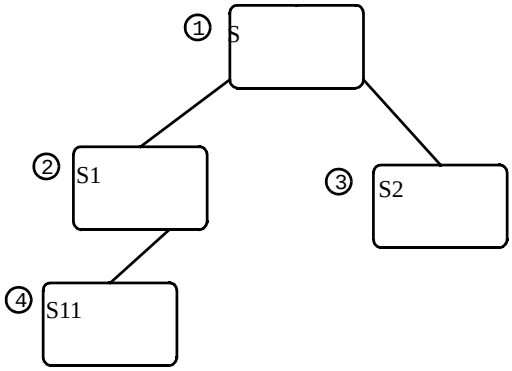
- `stm_success`
- `stm_nil_list`
- `stm_elements_not_hierarchical`

Example

Hierarchical elements in a chart can be ordered by *levels*. See the following statechart:



Hierarchically, the states can be drawn as shown as follows:



The set of elements, `{S1,S2}`, comprise a level. Assume you perform a “sort_by_level” function on the states in statechart `S`. The sorted order would be `S`, `S1`, `S2`, `S11`.

Note

The order of elements within the same level appear in an arbitrary order in the output. For example, `S2` might appear before `S1` because they are of the same level. However, the order of levels is “top-to-bottom” (here, `S` precedes `S1` or `S2`).

stm_list_sort_by_name

Function type: `LIST OF ELEMENT`

Description

Sorts the specified list of Statemate elements alphabetically by name.

Note the following:

- The function returns the status code `stm_elements_without_name` when you attempt to apply this function to a list that contains unnamed elements.
- The function receives and returns a list of element IDs, *not* a list of element names.

Syntax

```
stm_list_sort_by_name (list, status)
```

Arguments

Argument	Input/Output	Type	Description
`list`	In	LIST OF ELEMENT	The list of Statemate elements to be sorted. This input lists consists of element IDs.
`status`	Out	INTEGER	The function status code.

Status Codes

- `stm_success`
- `stm_nil_list`
- `stm_elements_without_name`

Example

Suppose you want to write a particular list of activities from the database to your document. You extract the activities of interest using single-element and query functions and build a list of such activities. This list is assigned to the variable `act_list`. To alphabetically sort the activities by their names, your template should contain the following statements:

```
VARIABLE
   LIST OF ACTIVITY   act_list, ord_act_list;
   ACTIVITY           activ;
   INTEGER            status;
                  .
                  .
ord_act_list := stm_list_sort_by_name (act_list, status);
WRITE ('\n Ordered list of activities:');
FOR activ IN ord_act_list LOOP
  WRITE ('\n', stm_r_ac_name (activ, status));
END LOOP;
```

stm_list_sort_by_synonym

Function type: `LIST OF ELEMENT`

Description

Sorts the specified list of Statemate elements alphabetically by their synonyms.

Note the following:

- The function returns the status code `stm_elements_without_name` when you attempt to apply this function to a list that contains unnamed elements.
- The function receives and returns a list of element IDs, *not* a list of element names.

Syntax

```
stm_list_sort_by_synonym (list, status)
```

Arguments

Argument	Input/Output	Type	Description
`list`	In	LIST OF ELEMENT	The list of Statemate elements to be sorted. This input lists consists of element IDs.
`status`	Out	INTEGER	The function status code.

Status Codes

- `stm_success`
- `stm_nil_list`
- `stm_elements_without_synonym`

Example

Suppose you want to write a particular list of activities from the database to your document. You extract the activities of interest using single-element and query functions and build a list of such activities. This list is assigned to the variable `act_list`. To alphabetically sort the activities by their synonyms, your template should contain the following statements:

```
VARIABLE
    LIST OF ACTIVITY        act_list, ord_act_list;
    MODULE                  act;
    INTEGER                 status;
                    .
                    .
                    .
ord_act_list = stm_list_sort_by_synonym (act_list,
    status);
WRITE ('\n Ordered list of activities:');
FOR act IN ord_act_list LOOP
   WRITE ('\n', stm_r_ac_synonym (act, status), '\t',
        stm_r_ac_name (act, status));
END LOOP;
```

stm_list_sort_by_type

Function type: `LIST OF ELEMENT`

Description

Sorts the specified list of Statemate elements by type.

Note that the function receives and returns a list of element IDs, *not* a list of element names.

Syntax

```
stm_list_sort_by_type (el_list, status)
```

Arguments

Argument	Input/Output	Type	Description
`el_list`	In	LIST OF ELEMENT	The list of Statemate element IDs to be sorted
`status`	Out	INTEGER	The function status code

Status Codes

- `stm_success`
- `stm_nil_list`

stm_multiline_to_one

Function type: `STRING`

Description

Converts the specified multiline string (with new lines) to a one-line string (without the new lines).

Syntax

```
stm_multiline_to_one (string)
```

Arguments

Argument	Input/Output	Type	Description
`string`	In	STRING	The multiline string

stm_multiline_to_strings

Function type: `LIST OF STRING`

Description

Converts the specified multiline expression to a list of strings.

Syntax

```
stm_multiline_to_strings (string)
```

Arguments

Argument	Input/Output	Type	Description
`string`	In	STRING	The multiline expression

stm_plot

Function type: `INTEGER`

Description

Generates a plot file with the indicated parameters, such as plot size, output device, and so on. The plot parameters are the same for all the different plot types (statecharts, activity charts, or module charts).

The output is designated for a particular device (one of the output devices defined in Statemate). The destination of the plot output is specified by one of the parameters. If its destination is not specified, the plot is included as part of the output segment file.

Syntax

```
stm_plot (id, plot_file, width, height, with_labels, with_names, with_notes,
device, date_position, title_position, title, do_rotate, with_file_header,
actual_height)
```

Example

Consider a template that contains the following statements:

```
VARIABLE
 CHART     ch_id;
 INTEGER        status;
 FLOAT          real_ht;
              .
ch_id:= stm_r_ch ('XL25', status);
stm_plot (ch_id, '/sam/p_xl25', 5.0, 7.0, true, true,
   false,          'POSTSCRIPT', stm_plt_top,
   'SystemXL25', true, true          , real_ht);
              .
```

stm_plot_hyper_exp

Function type: `INTEGER`

Description

Generates the hyperlinks in a sequence diagram.

Syntax

```
stm_plot_hyper_exp (id, plot_file, width, height, with_labels, with_names,
with_notes, with_hyperlink, device, date_position, title_position, title,
do_rotate, with_file_header, actual_height, with_breakpages, pages_in_x,
pages_in_y, page_index_in_x, page_index_in_y, with_hyperlink_exp)
```

Arguments

Argument	Input/ Output	Type	Description
`id`	In	Statemate element	The ID number of the Statemate chart to be plotted.
`plot_file`	In	STRING	The name of the file destination to which the plot is written. The operating system path name conventions are followed. You can specify a full path name to any directory for which you have write access. If you specify a simple file name, the plot is written to your workarea. If you do not specify a value (“), the plot is included as part of the output segment file.
`width`	In	FLOAT	The maximum possible width of the plot (in inches).
`height`	In	FLOAT	The maximum possible length of the plot (in inches). If you specify a plot size (width and height parameters) that is larger than the paper size defined for the specific printer, the plot simply uses the maximum allowable height and width defined for that printer.
`with_labels`	In	BOOLEAN	Determines whether labels are plotted (TRUE) or not (FALSE).
`with_names`	In	BOOLEAN	Determines whether names are plotted (TRUE) or not (FALSE).
`with_notes`	In	BOOLEAN	Determines whether notes are plotted (TRUE) or not (FALSE).
`with_hyperlink`	In	BOOLEAN	Specifies whether to generate hyperlinks for lifelines and referenced sequence diagrams (TRUE).

Argument	Input/ Output	Type	Description
`device`	In	STRING	Specifies the plotting device. This can be a supported formatting language if the plot is to be handled by a formatting processing system that has its own graphics language. To configure a new plotter or printer, select **Utilities > Output Devices** from the main Statemate window. Plots created using the Word format in the Output Devie diaglog are RTF files.
`date_position`	In	STRING	The position of the date. This is a string that indicates where to place the plot date. The possible values are as follows: • `stm_plt_none`—The date is not included. • `stm_plt_top`—The date is placed at the top of the plot. • `stm_plt_bottom`—The date is placed at the bottom of the plot.
`title_position`	In	STRING	The title position. This is a string that indicates where to place the plot title. The possible values are as follows: • `stm_plt_none`—The title is not included. • `stm_plt_top`—The title is placed at the top of the plot. • `stm_plt_bottom`—The title is placed at the bottom of the plot.
`title`	In	STRING	Specifies the title to be printed with the plot.
`do_rotate`	In	BOOLEAN	Determines whether the orientation of the plot is landscape (TRUE) or portrait (FALSE).
`with_file_header`	In	BOOLEAN	Indicates whether a header is added to the file (TRUE). Use this option if you do not want the plot as part of the document (FALSE).
`actual_height`	Out	FLOAT	Specifies the actual height (in inches) of the plotted output.
`with_breakpages`	In	BOOLEAN	Specifies whether to break the SD across multiple pages (true).
`pages_in_x`	Out	INTEGER	Specifies how many pages the tool attempted to break the SD into along the x-axis. Note that if `pages_in_x==0` and `pages_in_y==0`, the tool calculates a break pages scheme and assigns these variables so they can be read by the user after the call.

Argument	Input/ Output	Type	Description
`pages_in_y`	Out	INTEGER	Specifies how many pages the tool attempted to break the SD into along the y-axis.
`page_index_in_x`	In	INTEGER	Plots the *i*th page in the `x-axis`.
`page_index_in_y`	In	INTEGER	Plots the *i*th page in the `y-axis`.
`with_hyperlink_exp`	In	BOOLEAN	Specifies whether to generate hyperlinks for message labels (true).

Status Codes

- `stm_success`
- `stm_can_not_open_file`
- `stm_id_out_of_range`
- `stm_not_enough_memory`
- `stm_id_not_found`
- `stm_empty_chart`
- `stm_unknown_plotter`
- `stm_plot_failure`
- `stm_unresolved`
- `stm_illegal_parameter`

stm_plot_with_autonumber

Function type: `Integer`

Description

Prints a sequence diagram with numbered scenarios.

Syntax

```
stm_plot_with_autonumber (id, plot_file, width, height, with_labels,
with_names, with_notes, with_hyperlink, plot_type, date_position,
title_position, title, do_rotate, with_file_header, actual_height,
with_breakpages, pages_in_x, pages_in_y, page_index_in_x, page_index_in_y,
with_hyperlink_exp, with_headerline, headerline_y, with_autonumber)
```

Arguments

Argument	Input/ Output	Type	Description
`id`	In	Statemate element	The ID number of the Statemate chart to be plotted.
`plot_file`	In	STRING	The name of the file destination to which the plot is written. The operating system path name conventions are followed. You can specify a full path name to any directory for which you have write access. If you specify a simple file name, the plot is written to your workarea. If you do not specify a value (‘), the plot is included as part of the output segment file.
`width`	In	FLOAT	The maximum possible width of the plot (in inches).
`height`	In	FLOAT	The maximum possible length of the plot (in inches). If you specify a plot size (width and height parameters) that is larger than the paper size defined for the specific printer, the plot simply uses the maximum allowable height and width defined for that printer.
`with_labels`	In	BOOLEAN	Determines whether labels are plotted (true) or not (false).
`with_names`	In	BOOLEAN	Determines whether names are plotted (true) or not (false).
`with_notes`	In	BOOLEAN	Determines whether notes are plotted (true) or not (false).
`with_hyperlink`	In	BOOLEAN	Specifies whether to generate hyperlinks for lifelines and referenced sequence diagrams (true).

Argument	Input/ Output	Type	Description
plot_type	In	STRING	Specifies the plotting device. This can be a supported formatting language if the plot is to be handled by a formatting processing system that has its own graphics language. To configure a new plotter or printer, select **Utilities > Output Devices** from the main Statemate window. Plots created using the Word format in the Output Device dialog are RTF files.
date_position	In	STRING	The date position. This is a string that indicates where to place the plot date. The possible values are as follows: • stm_plt_none—The date is not included. • stm_plt_top—The date is placed at the top of the plot. • stm_plt_bottom—The date is placed at the bottom of the plot.
title_position	In	STRING	The title position. This is a string that indicates where to place the plot title. The possible vlues are as follows: • stm_plt_none—The title is not included. • stm_plt_top—The title is placed at the top of the plot. • stm_plt_bottom—The title is placed at the bottom of the plot.
title	In	STRING	Specifies the title to be printed with the plot.
do_rotate	In	BOOLEAN	Determines whether the orientation of the plot is landscape (true) or portrait (false).
with_file_header	In	BOOLEAN	Indicates whether a header is added to the file (true). Use this option if you do not want the plot as part of the document (false).
actual_height	Out	FLOAT	Specifies the actual height (in inches) of the plotted output.
with_breakpages	In	BOOLEAN	Specifies whether to break the SD across multiple pages (true).
pages_in_x	Out	INTEGER	Specifies how many pages the tool attempted to break the SD into along the x-axis. Note that if pages_in_x==0 and pages_in_y==0, the tool calculates a break pages scheme and assigns these variables so they can be read by the user after the call.
pages_in_y	Out	INTEGER	Specifies how many pages the tool attempted to break the SD into along the y-axis.
page_index_in_x	In	INTEGER	Plots the *i*th page in the x-axis.

Argument	Input/ Output	Type	Description
page_index_in_y	In	INTEGER	Plots the *i*th page in the y-axis.
with_hyperlink_exp	In	BOOLEAN	Specifies whether to generate hyperlinks for message labels (true).
with_headerline	In	BOOLEAN	Specifies whether to print the names of the lifelines on every page (true).
headerline_y	In	FLOAT	Defines the vertical coordinate on the page of the headerline. This is usually 1.0.
with_autonumber	In	BOOLEAN	Specifies whether to print the SD scenario numbers (true).

Status Codes

- stm_success
- stm_can_not_open_file
- stm_id_out_of_range
- stm_not_enough_memory
- stm_id_not_found
- stm_empty_chart
- stm_unknown_plotter
- stm_plot_failure
- stm_unresolved
- stm_illegal_parameter

stm_plot_with_break

Function type: `INTEGER`

Description

Breaks a sequence diagram across multiple pages.

Syntax

```
stm_plot_with_break (id, plot_file, width, height, with_labels, with_names,
with_notes, with_hyperlink, plot_type, date_position, title_position, title,
do_rotate, with_file_header, actual_height, with_breakpages, pages_in_x,
pages_in_y, page_index_in_x, page_index_in_y)
```

Arguments

Argument	Input/ Output	Type	Description
`id`	In	Statemate element	The ID number of the Statemate chart to be plotted.
`plot_file`	In	STRING	The name of the file destination to which the plot is written. The operating system path name conventions are followed. You can specify a full path name to any directory for which you have write access. If you specify a simple file name, the plot is written to your workarea.If you do not specify a value (‘’), the plot is included as part of the output segment file.
`width`	In	FLOAT	The maximum possible width of the plot (in inches).
`height`	In	FLOAT	The maximum possible length of the plot (in inches). If you specify a plot size (width and height parameters) that is larger than the paper size defined for the specific printer, the plot simply uses the maximum allowable height and width defined for that printer.
`with_labels`	In	BOOLEAN	Determines whether labels are plotted (TRUE) or not (FALSE).
`with_names`	In	BOOLEAN	Determines whether names are plotted (TRUE) or not (FALSE).
`with_notes`	In	BOOLEAN	Determines whether notes are plotted (TRUE) or not (FALSE).
`with_hyperlink`	In	BOOLEAN	Specifies whether to generate hyperlinks for lifelines and referenced sequence diagrams (TRUE).

Argument	Input/ Output	Type	Description
`plot_type`	In	STRING	Specifies the plot type. This can be a supported formatting language if the plot is to be handled by a formatting processing system that has its own graphics language. To configure a new plotter or printer, select **Utilities > Output Devices** from the main Statemate window. Plots created using the Word format in the Output Device dialog are RTF files.
`date_position`	In	STRING	The date position. This is a string that indicates where to place the plot date. The possible values are as follows: • `stm_plt_none`—The date is not included. • `stm_plt_top`—The date is placed at the top of the plot. • `stm_plt_bottom`—The date is placed at the bottom of the plot.
`title_position`	In	STRING	The title position. This is a string that indicates where to place the plot title. The possible values are as follows: • `stm_plt_none`—The title is not included. • `stm_plt_top`—The title is placed at the top of the plot. • `stm_plt_bottom`—The title is placed at the bottom of the plot.
`title`	In	STRING	Specifies the title to be printed with the plot.
`do_rotate`	In	BOOLEAN	Determines whether the orientation of the plot is landscape (TRUE) or portrait (FALSE).
`with_file_header`	In	BOOLEAN	Indicates whether a header is added to the file (TRUE). Use this option if you do not want the plot as part of the document (FALSE).
`actual_height`	Out	FLOAT	Specifies the actual height (in inches) of the plotted output.
`with_breakpages`	In	BOOLEAN	Specifies whether to break the SD across multiple pages (TRUE).
`pages_in_x`	Out	INTEGER	Specifies how many pages the tool attempted to break the SD into along the `x-axis`. Note that if `pages_in_x==0` and `pages_in_y==0`, the tool calculates a break pages scheme and assigns these variables so they can be read by the user after the call.

Argument	Input/ Output	Type	Description
pages_in_y	Out	INTEGER	Specifies how many pages the tool attempted to break the SD into along the y-axis.
page_index_in_x	In	INTEGER	Plots the *i*th page in the x-axis.
page_index_in_y	In	INTEGER	Plots the *i*th page in the y-axis.

Status Codes

- stm_success
- stm_can_not_open_file
- stm_id_out_of_range
- stm_not_enough_memory
- stm_id_not_found
- stm_empty_chart
- stm_unknown_plotter
- stm_plot_failure
- stm_unresolved
- stm_illegal_parameter

Notes

Function parameters are as follows:

```
boolean with_hyperlink (IN) /* generate hyperlinks */
boolean with_breakpages (IN) /* enable break pages */
integer pages_in_x (OUT) /* try to break to # of pages in x axis */
integer pages_in_y (OUT) /* try to break to # of pages in y axis */
```

If `pages_in_x == 0` and `pages_in_y==0`, the tool calculates a break pages scheme and assigns these variables so they can be read by the user after the call.

```
integer page_index_in_x (IN) /*plot the ith page in x axis */
integer page_index_in_y (IN) /*plot the ith page in y axis */
```

Call the function `STM_PLOT_SET_DATA()` before plotting a sequence diagram using `STM_PLOT_WITH_BREAK`. Call the function `STM_PLOT_RESET_DATA()` after finishing the sequence diagram multiple pages plot.

stm_plot_with_headerline

Function type: Integer

Description

Prints a sequence diagram with the names of lifelines on every page.

Syntax

```
stm_plot_with_headerline (id, plot_file, width, height, with_labels,
with_names, with_notes, with_hyperlink, plot_type, date_position,
title_position, title, do_rotate, with_file_header, actual_height,
with_breakpages, pages_in_x, pages_in_y, page_index_in_x, page_index_in_y,
with_hyperlink_exp, with_headerline, headerline_y,)
```

Arguments

Argument	Input/ Output	Type	Description
`id`	In	Statmate element	The ID number of the Statemate chart to be plotted.
`plot_file`	In	STRING	The name of the file destination to which the plot is written. The operating system path name conventions are followed. You can specify a full path name to any directory for which you have write access. If you specify a simple file name, the plot is written to your workarea. If you do not specify a value (‘), the plot is included as part of the output segment file.
`width`	In	FLOAT	The maximum possible width of the plot (in inches).
`height`	In	FLOAT	The maximum possible length of the plot (in inches). If you specify a plot size (width and height parameters) that is larger than the paper size defined for the specific printer, the plot simply uses the maximum allowable height and width defined for that printer.
`with_labels`	In	BOOLEAN	Determines whether labels are plotted (TRUE) or not (FALSE).
`with_names`	In	BOOLEAN	Determines whether names are plotted (TRUE) or not (FALSE).
`with_notes`	In	BOOLEAN	Determines whether notes are plotted (TRUE) or not (FALSE).
`with_hyperlink`	In	BOOLEAN	Specifies whether to generate hyperlinks for lifelines and referenced sequence diagrams (TRUE).

Argument	Input/ Output	Type	Description
`plot_type`	In	STRING	Specifies the plot type. This can be a supported formatting language if the plot is to be handled by a formatting processing system that has its own graphics language. To configure a new plotter or printer, select **Utilities > Output Devices** from the main Statemate window. Plots created using the Word format in the Output Device dialog are RTF files.
`date_position`	In	STRING	The date position. This is a string that indicates where to place the plot date. The possible values are as follows: • `stm_plt_none`—The date is not included. • `stm_plt_top`—The date is placed at the top of the plot. • `stm_plt_bottom`—The date is placed at the bottom of the plot.
`title_position`	In	STRING	The title position. This is a string that indicates where to place the plot title. The possible values are as follows: • `stm_plt_none`—The title is not included. • `stm_plt_top`—The title is placed at the top of the plot. • `stm_plt_bottom`—The title is placed at the bottom of the plot.
`title`	In	STRING	Specifies the title to be printed with the plot.
`do_rotate`	In	BOOLEAN	Determines whether the orientation of the plot is landscape (TRUE) or portrait (FALSE).
`with_file_header`	In	BOOLEAN	Indicates whether a header is added to the file (TRUE). Use this option if you do not want the plot as part of the document (FALSE).
`actual_height`	Out	FLOAT	Specifies the actual height (in inches) of the plotted output.
`with_breakpages`	In	BOOLEAN	Specifies whether to break the SD across multiple pages (TRUE).
`pages_in_x`	Out	INTEGER	Specifies how many pages the tool attempted to break the SD into along the `x`-axis. Note that if `pages_in_x==0` and `pages_in_y==0`, the tool calculates a break pages scheme and assigns these variables so they can be read by the user after the call.
`pages_in_y`	Out	INTEGER	Specifies how many pages the tool attempted to break the SD into along the y-axis.
`page_index_in_x`	In	INTEGER	Plots the *i*th page in the `x-axis`.
`page_index_in_y`	In	INTEGER	Plots the *i*th page in the `y-axis`.

Argument	Input/ Output	Type	Description
with_hyperlink_exp	In	BOOLEAN	Specifies whether to generate hyperlinks for message labels (TRUE).
with_headerline	In	BOOLEAN	Specifies whether to print the names of the lifelines on every page (TRUE).
headerline_y	In	FLOAT	Defines the vertical coordinate on the page of the headerline. This is usually 1.0.

Status Codes

- stm_success
- stm_can_not_open_file
- stm_id_out_of_range
- stm_not_enough_memory
- stm_id_not_found
- stm_empty_chart
- stm_unknown_plotter
- stm_plot_failure
- stm_unresolved
- stm_illegal_parameter

stm_replace_string

Function type: `STRING`

Description

Replaces the specified pattern with a new pattern within a string.

Syntax

```
stm_replace_string (orig_str, find_str, repl_str)
```

Arguments

Argument	Input/Output	Type	Description
`orig_str`	In	`STRING`	The string within which to search and replace
`find_str`	In	`STRING`	The pattern to search for
`repl_str`	In	`STRING`	The replacement pattern

stm_replace_word

Function type: `STRING`

Description

Replaces the specified word with a new word within a given string. A word has whitespace before and after it, preventing substitution where there is a partial match for the `find_str` parameter.

Syntax

```
stm_replace_word (orig_str, find_str, repl_str, delimiters, format,
replace_in_comment, is_identifier)
```

Arguments

Argument	Input/ Output	Type	Description
`orig_str`	In	`STRING`	The string within which to search and replace
`find_str`	In	`STRING`	The word to search for
`repl_str`	In	`STRING`	The replacement word
`delimiters`	In	`STRING`	A delimter to add whitespace
`format`	In	`STRING`	FrameMaker or Word
`replace_in_comment`	In	`BOOLEAN`	Specifies whether to replace the word in commented regions of the action language or user code
`is_identifier`	In	`BOOLEAN`	Specifies whether `find_str` is the name of a Statemate element

stm_set_rpt_formator

Function type: INTEGER

Description

Sets the formatter to use for Statemate reports. The possible values are FRAMEMAKER and WORD.

Syntax

```
stm_set_rpt_formator (formatter)
```

Arguments

Argument	Input/Output	Type	Description
`formatter`	In	INTEGER	The formatter to use for reports

stm_string_retain

Description:

When used in a DOC template, `STM_DISPOSE_MEMORY()` frees temporary memory used by the DOC system. If you want to avoid freeing a specific string, for example, general strings used in the template, call `STM_STRING_RETAIN(<str>)` on that string before calling `STM_DISPOSE_MEMORY()`.

Syntax

```
stm_string_retain(string)
```

Arguments

Argument	Input/ Output	Type	Description
`string`	In	STRING	The string to be retained in memory after performing `STM_DISPOSE_MEMORY()`

Status Codes

None

Example

```
STRING str;
STM_STRING_RETAIN(str)
```

stm_str_list_first_element

Function type: `STRING`

Description

Returns the first item in the specified list of strings.

Syntax

```
stm_str_list_first_element (list, status)
```

Arguments

Argument	Input/Output	Type	Description
`list`	In	LIST OF STRING	The list of strings
`status`	Out	INTEGER	The function status code

Status Codes

- `stm_success`
- `stm_nil_list`
- `stm_list_element_does_not_exist`

Example

Assume you have a list of strings `S1`, `S2`, `S3`, and `S4` assigned to the variable `str_list`. You locate the string `S1` by calling `stm_str_list_first_element`. `S1` becomes the current item. Your template should contain the following statements:

```
VARIABLE
    LIST OF STRING    str_list;
    STRING            str;
    INTEGER           status;
                  .
                  .
str = stm_str_list_first_element (str_list, status);
WRITE ('\n The first string in the list is:  ', str);
str = stm_str_list_next_element (str_list, status);
WRITE ('\n The second string in the list is: ', str);
                  .
                  .
```

stm_str_list_last_element

Function type: `STRING`

Description

Returns the element ID of the last item appearing in the specified list of strings.

Note the following:

- The first position in the string is 0.
- If the function fails, it returns an empty string.

Syntax

```
stm_str_list_last_element (list, status)
```

Arguments

Argument	Input/Output	Type	Description
`string`	In	`LIST OF STRING`	The list of strings
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_nil_list`
- `stm_list_element_does_not_exist`

Example

Assume there is a list of strings (`S1`, `S2`, `S3`, and `S4`) assigned to the variable `str_list`. You locate the string `S4` by calling `stm_str_list_last_element`. `S4` becomes the "current" item. Your template should contain the following statements:

```
VARIABLE
    LIST OF STRING   str_list;
    STRING           str;
    INTEGER          status;
    .
    .
str := stm_str_list_last_element (str_list, status);
WRITE ('\n The last string in the list is:  ', str);
str := stm_str_list_previous_element (str_list, status);
WRITE ('\n The third string in the list is: ', str);
    .
    .
```

stm_str_list_length

Function type: INTEGER

Description

Returns the number of items in the specified list of strings.

Syntax

```
stm_str_list_length (list, status)
```

Arguments

Argument	Input/Output	Type	Description
`list`	In	`LIST OF STRING`	The list of strings whose length you want
`status`	Out	INTEGER	The function status code

Status Codes

- `stm_success`
- `stm_nil_list`

Example

Assume you have extracted all the static reactions in state `ST1` from the database. Before writing the list of strings to your document, you want to make sure that it will not span more than 30 lines of text (one page length). Your template should contain the following statements:

```
VARIABLE
    STATE            st;
    STRING           str;
    LIST OF STRING   str_list;
    INTEGER          str_list_len, status;
    CONSTANT INTEGER page_len :=  30;
                  .
                  .
st:=stm_r_st ('ST1',status);
str_list:=stm_r_st_static_reactions (st, status);

str_list_len := stm_str_list_length (st, status);
IF str_list_len < page_len
   WRITE ('\n List of Reactions: ');
FOR str IN str_list LOOP
     WRITE ('\n', str);
END LOOP;
END IF
```

stm_str_list_next_element

Function type: `STRING`

Description

Returns the next item in the specified list of strings.

Note that “next” refers to the item physically located after the current item in the list of strings. The “current” item is determined using the utility function `stm_str_list_first_element`.

Syntax

```
stm_str_list_next_element (list, status)
```

Arguments

Argument	Input/Output	Type	Description
`list`	In	`LIST OF STRING`	The list of strings
`status`	Out	INTEGER	The function status code

Status Codes

- `stm_success`
- `stm_nil_list`
- `stm_list_element_does_not_exist`

Example

Assume we have a list of strings `S1`, `S2`, `S3`, and `S4` assigned to the variable `str_list`. You locate the string `S1` by calling `stm_str_list_first_element`. `S1` becomes the current item. To find the next element in the list, use the following statements:

```
VARIABLE
    LIST OF STRING   str_list;
    STRING           str;
    INTEGER          status;
                .
                .
str = stm_str_list_first_element (str_list, status);
WRITE ('\n The first string in the list is:  ', str);
str := stm_str_list_next_element (str_list, status);
WRITE ('\n The second string in the list is: ',str);
              .
              .
```

This function is often used in loop statements.

stm_str_list_previous_element

Function type: `STRING`

Description

Returns the previous item in the specified list of strings.

Note that “previous” refers to the item physically located before the current item in the list of strings. The “current” item is determined using the utility function `stm_str_list_last_element`.

Syntax

```
stm_str_list_previous_element (list, status)
```

Arguments

Argument	Input/Output	Type	Description
`list`	In	`LIST OF STRING`	The list of strings
`status`	Out	INTEGER	The function status code

Status Codes

- `stm_success`
- `stm_nil_list`
- `stm_list_element_does_not_exist`

Example

Assume you have a list of strings `S1, S2, S3,` and `S4` assigned to the variable `str_list`. You locate the string `S4` by calling `stm_str_list_last_element`. `S4` becomes the current item. Your template should contain the following statements:

```
VARIABLE
    LIST OF STRING  str_list;
    STRING          str;
    INTEGER         status;
                .
                .
str := stm_str_list_last_element (str_list, status);
WRITE ('\n The last string in the list is:  ', str);
str := stm_str_list_previous_element (str_list, status);
WRITE ('\n The third string in the list is: ', str);
                .
                .
```

This function is often used in loop statements.

stm_str_list_to_str

Function type: `STRING`

Description

Returns the string representation of the specified list of strings.

Syntax

```
stm_str_list_to_str (s_list)
```

Arguments

Argument	Input/Output	Type	Description
`s_list`	In	`LIST OF STRING`	The list of strings to be converted

Status Codes

- `stm_success`

stm_str_to_list

Function type: `LIST OF STRING`

Description

Converts the specified string to a list of strings.

Syntax

```
stm_str_to_list (s, delimiter)
```

Arguments

Argument	Input/Output	Type	Description
`s`	In	`STRING`	The string to convert
`delimiter`	In	`STRING`	A delimiter for whitespace

stm_string_extract

Function type: `STRING`

Description

Extracts a portion of the specified string. The extracted portion starts at the `index` position and its length is `len`.

Syntax

```
stm_string_extract (string, index, len, status)
```

Arguments

Argument	Input/Output	Type	Description
`string`	In	`STRING`	The entire string
`index`	In	`INTEGER`	The starting point of the extraction
`len`	In	`INTEGER`	The number of characters to extract from the string
`status`	Out	`INTEGER`	The function status code

Status Codes

- `stm_success`
- `stm_null_string`
- `stm_illegal_index`
- `stm_illegal_len`

Example

```
stm_string_extract ('ABCDE',0,2,status) = 'AB'
stm_string_extract ('ABCDE',3,2,status) = 'DE'
stm_string_extract ('ABCDE',0,7,status) = ''
    and status = stm_illegal_len
```

stm_string_free

Function type: `INTEGER`

Description

Frees the memory used by the specified string.

Syntax

```
stm_string_free (s)
```

Arguments

Argument	Input/Output	Type	Description
s	In	`STRING`	The string whose memory you want to free

stm_string_retain

Description:

When used in a DOC template, `STM_DISPOSE_MEMORY()` frees temporary memory used by the DOC system. If you want to avoid freeing a specific string, for example, general strings used in the template, call `STM_STRING_RETAIN(<str>)` on that string before calling `STM_DISPOSE_MEMORY()`.

Syntax

```
stm_string_retain(string)
```

Arguments

Argument	Input/ Output	Type	Description
`string`	In	`STRING`	The string to be retained in memory after performing `STM_DISPOSE_MEMORY()`

Status Codes

None

Example

```
STRING str;
STM_STRING_RETAIN(str0
```

stm_string_to_int

Function type: `INTEGER`

Description

Returns an integer value of a decimal string representation of a number.

Syntax

```
stm_string_to_int (string)
```

Arguments

Argument	Input/Output	Type	Description
`string`	In	`STRING`	The string to be converted

Status Codes

- `stm_success`

stm_strlen

Function type: `INTEGER`

Description

Returns the length of the specified string.

Syntax

```
stm_strlen (string)
```

Arguments

Argument	Input/Output	Type	Description
`string`	In	`STRING`	The string whose length you want

Status Codes

- `stm_success`

Example

```
len := stm_strlen ('ABC');
len is 3.
```

stm_trigger_of_reaction

Function type: `STRING`

Description

Returns the trigger part of a reaction (label of transition or static reaction). The syntax of the reaction is `trigger/action`.

Note the following:

- The reaction is achieved by the following single-element functions:
  - `stm_r_st_reactions`
  - `stm_r_tr_labels`
- The function returns an empty string when the trigger is missing.

Syntax

```
stm_trigger_of_reaction (reaction, status)
```

Arguments

Argument	Input/Output	Type	Description
`reaction`	In	`STRING`	The reaction to decompose
`status`	Out	`STRING`	The function status code

Status Codes

- `stm_success`

Example

To list all events that have influence on `S1`, which has several static reactions, use the following statements in your template:

```
Variable
    STATE             st_id;
    INTEGER           status;
    LIST OF STRING    reactions;
    STRING            rct;
st_id:=stm_r_st('S1',status);
reactions:=stm_r_st_reactions (st_id, status);
WRITE ('\n Triggers of reaction is S1:');
FOR rct IN reactions LOOP
    WRITE ('\n', stm_trigger_of_reaction (rct, status));
END FOR;
```

Project Management

This appendix describes special project management functions. For each function, the following information is provided:

- Description
- Syntax
- Arguments
- Status codes

The following table lists the project management functions.

Function	Description
stm_r_pm_member_workareas	Returns the workareas of the specified user
stm_r_pm_operator_projects	Returns a list of all the projects in which the specified user is a member
stm_r_pm_project_databank	Returns the databank name of the specified project in the project management database
stm_r_pm_project_manager	Returns the manager of the specified project in the project management database
stm_r_pm_project_members	Returns a list of all the members of the specified project in the project management database
stm_r_pm_projects	Returns a list of all the projects in the project management database

stm_r_pm_member_workareas

Function type: `LIST OF STRING`

Description

Returns the workareas of the specified user.

Syntax

```
stm_r_pm_member_workareas (o_name, p_name, status)
```

Arguments

Argument	Input/Output	Type	Description
oname	In	STRING	The name of the user
pname	In	STRING	The name of the project
status	Out	INTEGER	The function status code

Status Codes

- `stm_success`
- `stm_nonexistent_project`
- `stm_not_member_of_project`

stm_r_pm_operator_projects

Function type: `LIST OF STRING`

Description

Returns a list of all the projects in which the specified user is a member.

Syntax

```
stm_r_pm_operator_projects (oname, status)
```

Arguments

Argument	Input/Output	Type	Description
oname	In	STRING	The name of the user
status	Out	STRING	The function status code

Status Codes

- `stm_success`
- `stm_no_projects`

stm_r_pm_project_databank

Function type: `STRING`

Description

Returns the databank name of the specified project in the project management database.

Syntax

```
stm_r_pm_project_databank (pname, status)
```

Arguments

Argument	Input/Output	Type	Description
pname	In	STRING	The name of the project
status	Out	INTEGER	The function status code

Status Codes

- `stm_success`
- `stm_nonexistent_project`

stm_r_pm_project_manager

Function type: `STRING`

Description

Returns the manager of the specified project in the project management database.

Syntax

```
stm_r_pm_project_manager (pname, status)
```

Arguments

Argument	Input/Output	Type	Description
pname	In	STRING	The name of the project
status	Out	INTEGER	The function status code

Status Codes

- `stm_success`
- `stm_nonexistent_project`

stm_r_pm_project_members

Function type: `LIST OF STRING`

Description

Returns a list of all the members of the specified project in the project management database.

Syntax

```
stm_r_pm_project_members (pname, status)
```

Arguments

Argument	Input/Output	Type	Description
pname	In	STRING	The name of the project
status	Out	INTEGER	The function status code

Status Codes

- `stm_success`
- `stm_nonexistent_project`

stm_r_pm_projects

Function type: `LIST OF STRING`

Description

Returns a list of all the projects in the project management database.

Syntax

```
stm_r_pm_projects (status)
```

Arguments

Argument	Input/Output	Type	Description
status	Out	INTEGER	The function status code

Status Codes

- `stm_success`
- `stm_no_projects`

Function Status Codes

Status codes have three severity levels:

- **S** for success
- **W** for warning
- **E** for error

When a warning or error status is returned, attempts to execute statements using the return value of the function can produce erroneous or unexpected results. Therefore, you should check the return status codes to ensure that your function call is successful before using the returned values.

The following table lists the status codes and their severity levels.

Code	Status Name	Severity Level
–4	`stm_no_stm_root` UNIX: The `STM_ROOT` environment variable is not defined. VMS: The `STM$ROOT` or `STM$PM` logical name does not exist.	E
–3	`stm_obsolete_function` Irrelevant function for the current version.	E
–2	`stm_missing_elements_in_list` Input elements do not exist in the database.	W
–1	`stm_list_type_mismatch` Incorrect element type used in the query.	E
0	`stm_success` The function call was successful.	S
1	`stm_id_out_of_range` The specified ID is not valid for this element type.	E
2	`stm_id_not_found` An element with the specified ID does not exist.	E
3	`stm_illegal_name` The specified name is not legal.	E
4	`stm_name_not_found` The specified name does not exist.	E

5	`stm_name_not_unique` There is more than one element with the specified name, so a specific path name is required.	E
6	`stm_missing_name` The specified element has no name.	W
7	`stm_missing_synonym` The specified element has no synonym.	W
8	`stm_missing_short_description` The specified element has no short description.	W
9	`stm_missing_long_description` The specified element has no long description.	W
10	`stm_attribute_name_not_found` The specified element has no attribute name.	W
11	`stm_starting_keyword_not_found` The long description of the specified element does not contain the given starting keyword.	W
12	`stm_ending_keyword_not_found` The long description of the specified element does not contain the given ending keyword.	W
13	`stm_primitive_element` The element is primitive.	W
14	`stm_can_not_open_file` The operating system cannot open the file with the specified name.	E
15	`stm_illegal_address` The pointer address is illegal.	E
16	`stm_not_an_and_state` This state is not supposed to contain and-lines.	W
17	`stm_no_and_lines_in_and_state` This and-state is missing and-lines.	E
18	`stm_missing_graphic_data` Graphic data is missing from the element.	E
19	`stm_nil_list` There is no input list.	E
20	`stm_list_element_does_not_exist` The specified element does not exist.	W
21	`stm_cannot_compare_lists` The lists cannot be compared because the list types are different, or the lists are not initialized.	E
22	`stm_elements_without_name` The list cannot be sorted because its elements have no names.	E
23	`stm_elements_without_synonym` The list cannot be sorted because its elements have no synonyms.	E

24	`stm_elements_not_hierarchical` The list cannot be sorted because it is not hierarchical.	E
25	`stm_illegal_extract_type` You cannot extract this element type.	E
26	`stm_no_such_list` No such list exists	E
27	`stm_not_diagram_connector` There is no value in a connector that is not a diagram connector.	E
28	`stm_implicit_element` The element is defined implicitly—it is an internally defined entity.	E
29	`stm_missing_label` The element has no label.	W
30	`stm_unknown_plotter` The plotter type is unknown.	E
31	`stm_unresolved` The element is unresolved.	W
32	`stm_elements_without_attributes` The list cannot be sorted because its elements have no attributes.	E
33	`stm_not_instance` The element is not an instance.	E
34	`stm_no_updated_pmdb` The workarea database is not updated to the current version.	E
35	`stm_no_updated_projdb` The installation database is not updated to the current version.	E
36	`stm_no_legal_operator` The user is not authorized as a Statemate operator.	E
37	`stm_deadlock` Deadlock situation.	E
38	`stm_not_member_of_project` The user is not a member of the specified project.	E
39	`stm_nonexistent_project` The specified project does not exist.	E
40	`stm_not_enough_memory` The plot cannot be produced because there is not enough memory.	E
41	`stm_empty_chart` The plot file cannot be produced because the chart is empty.	E
42	`stm_plot_failure` The plot file was not produced because of a system error.	E
43	`stm_no_file_of_licensed_host` The file containing the name of the licensed host does not exist.	E

44	`stm_empty_file_of_licensed_host` The file containing the name of the licensed host is empty.	E
45	`stm_cannot_chdir_to_work_area` Could not change directory to the workarea.	E
46	`stm_cannot_write_to_file` No space is left on device for writing a file.	E
47	`stm_illegal_parameter` An illegal parameter value was supplied.	E
50	`stm_null_string` The input string is null.	E
51	`stm_illegal_len` The length value is illegal.	E
52	`stm_illegal_index` The index value is illegal.	E
53	`stm_cannot_read_file` Cannot read from a file that was not opened.	E
54	`stm_end_of_file` Reached the end-of-file.	E
55	`stm_not_a_parameter` The specified ID is not a parameter.	E
56	`stm_param_not_compatible` The actual and formal parameters are not compatible.	W
57	`stm_error_in_file` There is an error in the requirement file.	E
58	`stm_missing_field` A field is missing in the requirement record.	W
59	`stm_missing_user_type` The specified element has no user-defined type.	E
61	`stm_illegal_attribute_value` The attribute value is too long.	E
62	`stm_duplicate_attribute_pair` The specified attribute name/value pair already exists.	E
63	`stm_not_in_rw_transaction` Attempt to modify the database when not in a read/write transaction.	E
64	`stm_missing_of_enum_type` The specified element has no enumerated type associated with its array type definition.	W
65	`stm_missing_user_code` The specified element has no user code.	W
66	`stm_missing_subroutine_params` The specified element has no subroutine parameters.	W

67	`stm_missing_local_data` The specified element has no local data.	W
68	`stm_missing_global_data` The specified element has no global data.	W
69	`stm_no_connected_chart` The specified element is not connected to a chart.	W
70	`stm_attribute_cannot_be_deleted` The specified element's attribute cannot be deleted.	E
71	`stm_missing_cbk_binding` The specified element has no callback binding.	W
72	`stm_missing_subroutine_binding` The specified element has no subroutine binding.	W
73	`stm_missing_statemate_action_lang` The specified element has no action language.	W
74	`stm_no_projects` There are no projects in the project management database.	W
121	`stm_illegal_expression_n_chart` There is an illegal expression in the loaded chart.	E
122	`stm_error_in_chart` There is an error in the loaded chart.	E
123	`stm_cannot_open_chart_file` Cannot open the chart file to be loaded.	E
124	`stm_exceeded_max_id_number` There are more than 1023 IDs in the workarea.	E
125	`stm_chart_not_in_database` Cannot find a chart in the database to be saved or unloaded.	E
126	`stm_file_not_in_work-area` Cannot find a file in the workarea to be saved or unloaded.	E
127	`stm_cannot_copy_file` Cannot copy a file during a save or load operation.	E
128	`stm_cannot_create_file` Cannot create an auxiliary file during a load to the workarea.	E
129	`stm_illegal_version` An illegal version was specified for the load operation.	E
130	`stm_file_not_found` Cannot find a source file in the load operation.	E
131	`stm_not_loaded_because_modified` A modified version of loaded chart or file exists in the workarea.	E
132	`stm_not_loaded_because_new` A new version of the loaded chart or file exists in the workarea.	E

133	`stm_not_unloaded_modified` The chart or file to be unloaded is modified.	E
134	`stm_not_unloaded_new` The chart or file to be unloaded is new.	E
135	`stm_chart_is_active` The chart to be unloaded is currently being edited by a graphics editor.	E
136	`stm_error_in_save_operation` There was a write to disk error during the save operation.	E
137	`stm_illegal_load_mode` An illegal mode was specified for the load operation.	E
138	`stm_not_loaded_because_type` A chart with the same name, but of another type, exists in the workarea.	E
139	`stm_illegal_type` An illegal type of configuration item was specified.	E
140	`stm_illegal_parameters` An illegal parameter to the load function was specified.	E
141	`stm_illegal_bindings` There is an error in the loaded chart file.	E
142	`stm_too_long_line` There is a line too long in the loaded chart file.	E
143	`stm_instance_type_conflict` There is an instance type conflict in the loaded chart file.	E
144	`stm_usage_conflict` There is a usage conflict in the loaded chart file.	E
145	`stm_unrecognized_format` The loaded chart file contains an unrecognized conflict.	E
146	`stm_double_chart_parameters` There is an error in the loaded chart file.	E
147	`stm_double_chart_bindings` There is an error in the loaded chart file.	E

DGL Reserved Words

The following table lists the keywords reserved for specific DGL use. Do not use these words as names in your document templates.

ACTION	FIELD	OTHERWISE
ACTIVITY	FILE	OUTPUT
AND	FIRST	PARAMETER
ANY	FLOAT	PROCEDURE
ARRAY	FOR	PROGRAM
A_FLOW_LINE	FUNCTION	READ
BEGIN	IF	RETURN
BOOLEAN	IN	SEGMENT
CLOSE	INCLUDE	SELECT
CONDITION	INFORMATION_FLOW	STATE
CONNECTOR	INPUT	STOP
CONSTANT	INTEGER	STRING
DATA_ITEM	LIST	TEMPLATE
DATA_STORE	LOOP	THEN
DATA_TYPE	MODULE	TRANSITION
ELEMENT	M_FLOW_LINE	TRUE
ELSE	NOT	TYPE
END	NULL	VARIABLE
EVENT	OF	WHEN
EXECUTE	OPEN	WHILE
EXIT	OR	WRITE
FALSE		

BNF Syntax

This section lists the conventions for a widely used notational scheme for formal languages known as BNF, which stands for Bakus-Naur Form (formerly Bakus Normal Form). BNF was introduced in 1963 as a technique for defining programing languages.

In the Statemate documentation, a variation of the BNF notation is used to formally describe the DGL statements.

BNF Structure and Conventions

BNF grammar follows the following general structure:

```
nonterminal_symbol ? terminal_and/or_nonterminal_
    symbols
```

For example:

```
write_expression ? numeric_expression |string_expression
```

Symbols are delimited by spaces; underscores are frequently used for longer names.

Symbol Types

Terminal symbols are basic symbols that are not parsed further to derive their meaning. *Nonterminal symbols* can be further broken down by parsing.

Examples of terminal symbols are:

- Integer numbers intrinsically recognized as a numerical value
- Language keywords recognized by the system as representing some particular operation or function.

In **BNF for DGL Statements**, terminal symbols that are written exactly as they appear (for example, keywords of Statemate), are shown in all uppercase. Non-alphabetic characters not belonging to the BNF notation are also part of the syntax:

- Nonterminal symbols are written in lowercase or mixed case letters.
- Nonterminal symbols that are self-evident are not broken down further.

BNF Notations

The | indicates a mutually exclusive choice between symbols in a nonterminal symbol definition. For example:

```
variable_name | numeric_constant |integer | function_name
```

The ? separates the nonterminal symbol on the left from its definition on the right and can be read as “is defined as...”. For example:

```
relational_operator ? = | < > | < | <= | >=
```

Square brackets ([]) indicate that the symbols within the brackets are optional. For example:

```
[directory_name] filename
```

Curly braces ({}) indicate that the symbols within the braces are optional and can be repeated. For example:

```
begin {statement} end;
```

BNF for DGL Statements

This section lists the formal syntax (in BNF) for DGL. For ease of use, the DGL statements are presented in alphabetical order.

DGL Statement	BNF
abs_integer_literal	{digit}
abs_numeric_literal	abs_integer_literal | abs_real_literal
abs_real_literal	{digit}. [{digit}] | [{digit}]. {digit}
alpha_letter	A to Z, a to zy
assignment_statement	variable := expression;
boolean_expression	boolean_term | boolean_expression OR boolean_term
boolean_function_call	function_call
boolean_identifier	boolean_parameter | boolean_constant | boolean_variable
boolean_literal	TRUE | FALSE
boolean_primary	boolean_literal | boolean_identifier | expression relation_operation expression | NOT boolean_primary | boolean_function_call | (boolean_expression)

boolean_term	boolean_primary | boolean_term AND boolean_primary
calling_sequence	string_expression
constant_declaration	CONSTANT {const_object_specification}
const_ident_spec	identifier :=literal
const_ident_spec_list	const_ident_spec [{,const_ident_spec}]
constant_object_specification	type const_ident_spec_list
const_var_declaration	constant declaration I variable declaration
control_flow_statement	if_statement | select_statement | for_statement | while_statement | exit_statement | stop_statement
copy_text_statement	/@ text @/
close_statement	CLOSE (file_identifier);
element_expression	element_variable | element_function_call
element_function_call	function_call
execute_call	EXECUTE (calling_sequence);
exit_statement	EXIT;
expression	integer_expression | real_expression | boolean_expression | string_expression | element_expression | list_expression
file_description	file_name | file_identifier
file_identifier	file_variable
file_name	string_expression
file_statement	open_statement | close_statement | read_statement
for_statement	FOR variable IN list_expression LOOP statements END LOOP;
function_call	id (expressions [{, expression}])
global_par	[parameter_declaration] [const_var_declaration] BEGIN [statements] END;
ident_spec	identifier [:= literal]
ident_spec_list	ident_spec [{,ident_spec}]
identifier	alpha_letter [{identifier_letter}]
identifier_letter	A to Z, a to z, 0 to 9, and _
if_statement	IF boolean_expression THEN statements [ELSE statements] END IF;

include_statement	INCLUDE (file_description [,integer_variable]);
integer_expression	numeric_expression
integer_identifier	integer_parameter | integer_constant | integer_variable
integer_literal	[+ | -]abs_integer_literal
list_component integer_expression	expression | integer_expression..
list_component_literal	literal | integer_literal.. integer_literal
list_expression	list_term | & | list_expression + list_term | list_expression - list_term | list_expression & list_term
list_identifier	list_parameter | list_constant | list_variable
list_of_components	slist_component ["{", list_component"} See Note.
list_term	list_literal | list_identifier | "{" list_of_components "}" 1| list_term * list_term | (list_expression)
literal	integer_literal | real_literal | boolean_literal | string_literal | list_literal
numeric_expression	numeric_term | numeric_expression + numeric_term| numeric_expression - numeric_term
numeric_factor	numeric_factor ** numeric_primary| numeric_primary
numeric_function_call	function_call
numeric_identifier	integer_identifier | real_identifier
numeric_primary	abs_numeric_literal | numeric_identifier | nurneric_function_call | (numeric_expression)
numeric_term	numeric_factor | numeric_term * numeric_factor | numeric_term / numeric_factor | + numeric_factor | -- numeric_factor |
object_specification	type ident_spec_list;
open_mode	INPUT I OUTPUT
open_statement	OPEN (file_identifier, file_name, open_mode[,integer_variable]);

output_statement	copy_text_statement | write_statement | include_statement | special_procedure_call | file_statement
parameter_declaration	PARAMETER object_specification
plot_call	function_call

procedure (call)	PROCEDURE procedure_name [return type];
	[{parameter_object_declaration}]
	[{variable_declaration}]
	BEGIN
	[statements]
	END;

procedure (name)	procedure proc_name
read_statement	READ (file_identifier {, simple_variable})
real_expression	numeric_expression
real_identifier	real_parameter | real_constant | real_variable
real_literal	[+ | -]abs_real_literal
relation_operation	= | < > | >= | > | < = | <
report_call	function_call

segment	SEGMENT segment_name;
	[{const_var declaration}]
	BEGIN
	[statements]
	END;

segment_name	identifier
select_statement	SELECT [FIRST | ANY] [{when_construct_with_any}] when_construct [OTHERWISE = > statements] END SELECT;
simple_type	INTEGER | REAL | BOOLEAN | STRING | FILE statemate_element
simple_variable	integer_variable | real_variable | string_variable
special_procedure_call	execute_call | report_call | plot_call | table_call

statemate_element	ELEMENT | STATE | ACTIVITY | DATA_STORE | MODULE | TRANSITION | A_FLOW_LINE | M_FLOW_LINE | EVENT| DATA_TYPE_FIELD | CONDITION | CONNECTOR | DATA_ITEM | ACTION | INFORMATION_FLOW
statement	assignment_statement | output_statement | control_flow_statement |
stop_statement	STOP;
string_expression	string_literal | string_identifier | string_function_call | string_expression + string_expression
string_function_call	function_call
string_identifier	string_parameter | string_constant | string_variable
string_literal	’{\n|\t|printable character}’
structured_type	LIST OF simple_type
table_call	function_call
template	template_header global_part {segment} {procedure}
template_header	TEMPLATE template_name;
template_name	identifier
type	simple_type | structured_type
variable_declaration	VARIABLE {object specification}
when_construct	{when sentence} [WHEN ANY => statements]
when_construct_with_any	{when sentence} [WHEN ANY => statements]
when_sentence	WHEN boolean_expression => statement
while_statement	[WHILE boolean_expression] LOOP statements END LOOP;
write_expression	expression [:integer_expression]
write_statement	WRITE ([file_identifier,] {write_expression});

Note

The quotation marks indicate that the braces here are not BNF notation, but are the actual brace characters themselves.

Index

E

F

N

O

P

Q

R

S

T

U

V

W