Software through Pictures®
Core

Release 7.1

Query and Reporting System

Software through Pictures Core Query and Reporting System

Release 7.1

Part No. 10-MN102/ST7100-01298/001

December 1998

Aonix reserves the right to make changes in the specifications and other information contained in this publication without prior notice. In case of doubt, the reader should consult Aonix to determine whether any such changes have been made. The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

Trademarks

World Headquarters

595 Market Street, 12th Floor
San Francisco, CA 94105
Phone: (800) 97-AONIX
Fax: (415) 543-0145
E-mail: info@aonix.com
http://www.aonix.com

Table of Contents

Chapter 3 The StP Query and Reporting Language

Chapter 5 Creating Documents

Preface

Software through Pictures (StP) is a family of multi-user integrated environments that supports the software development process, as well as maintenance and re-engineering of existing systems.

This manual is part of a complete StP documentation set of "Core" manuals, which includes: *Fundamentals of StP*, *Customizing StP*, *Query and Reporting System*, *Object Management System*, and *StP Administration*. Basic information about the StP user interface is documented in *Fundamentals of StP*.

The StP Query and Reporting System (QRS) enables you to use scripts written in the Query and Reporting Language (QRL) to extract information from the repository. You can perform operations on the information such as checking, including it in documents, and using it to generate code or schema.

The subjects covered in this manual include:

- Using the Script Manager
- Writing QRL scripts
- Using Object Management System (OMS) queries in QRL scripts
- Creating and formatting documents

Intended Audience

The *Query and Reporting System* manual is intended for users who have some understanding of the StP environment (its editors and repository) and their publishing tool. You should know how to use:

- Your StP environment
- Windows NT
- Your publishing tool: FrameMaker or Microsoft Word

Typographical Conventions

This manual uses the following typographical conventions:

Table 1: Typographical Conventions

Convention	Meaning
Palatino bold	Identifies commands**.**
`Courier`	Indicates system output and programming code.
`Courier bold`	Indicates information that must be typed exactly as shown, such as command syntax.
italics	Indicates pathnames, filenames, and ToolInfo variable names.
<angle brackets>	Surround variable information whose exact value you must supply.
[square brackets]	Surround optional information.

Contacting Aonix

You can contact Aonix using any of the following methods.

Reader Comments

Aonix welcomes your comments about its documentation. If you have any suggestions for improving *Query and Reporting System*, you can send email to docs@aonix.com.

Technical Support

If you need to contact Aonix Technical Support, you can do so by using the following email aliases:

Table 2: Technical Support Email Aliases

Country	Email Alias
Canada	support@aonix.com
France	customer@aonix.fr
Germany	stp-support@aonix.de
United Kingdom	stp-support@aonix.co.uk
United States	support@aonix.com

Users in other countries should contact their StP distributor.

Websites

You can visit us at the following websites:

Table 3: Aonix Websites

Country	Website URL
Canada	http://www.aonix.com
France	http://www.aonix.fr
Germany	http://www.aonix.de
United Kingdom	http://www.aonix.co.uk
United States	http://www.aonix.com

Related Reading

Query and Reporting System is part of a set of Software through Pictures documentation. For more information about StP Core and related subjects, refer to the sources listed in Table 4.

Table 4: Further Reading

For Information About	Refer To
Using the StP Desktop	*Fundamentals of StP*
Using standard StP editors and tools	*Fundamentals of StP*
Printing diagrams, tables, and reports	*Fundamentals of StP*
Internal components of StP and how to customize an StP installation	*Customizing StP*

Table 4: Further Reading (Continued)

For Information About	Refer To
StP Object Management System (OMS)	*Object Management System*
Installing StP on Windows NT	*StP Installation Guide*
Interacting directly with the StP storage manager, Sybase SQL Server	*StP Guide to Sybase SQL Server*
Managing an StP installation	*StP Administration*
Specific StP products information	The documentation provided with your StP product
Managing the database	The documentation provided with your relational database management system and *StP Administration*

1 Introduction to the StP Query and Reporting System

This chapter introduces the Software through Pictures Query and Reporting System. It describes:

The Software through Pictures Query and Reporting System (QRS) enables you to use scripts to retrieve information from the StP repository.

When you develop a system using StP, you store information about the system in the StP repository. Such information includes descriptions of specific symbols in a model, the relationships between symbols, and annotations, such as those used for code generation. Once system information is in the repository, you can use the QRS to:

- Read repository information
- Generate formatted documents that include text, diagrams, and tables
- Generate code (including C++, Java, Ada, IDL, and TOOL)

Components of QRS

Software through Pictures provides two ways of using the QRS:

- The Script Manager
- The Query and Reporting Language (QRL)

You can use the QRS in conjunction with the Object Management System (OMS) and format files to produce output for certain publishing products.

The Script Manager

The Script Manager, available from the StP Desktop, is the main graphical user interface to the QRS. You use it to run, modify, copy, and create scripts, as well as to modify their external variables' values.

For details, see Chapter 2, "The Script Manager."

The StP Query and Reporting Language

To access the repository, you use scripts written in the Query and Reporting Language (QRL), a C-like interpreted script language. You can write your own QRL scripts or use scripts supplied with StP.

QRL only reads data from the repository; it does not write to the repository.

For details regarding QRL syntax, see Chapter 3, "The StP Query and Reporting Language." For a list of all QRL built-in functions, see Appendix A, "Built-in Functions."

The OMS Query Language and QRL

You use OMS queries in QRL scripts to extract information from the repository.

For details on how to incorporate OMS queries in QRL scripts, see Chapter 4, "The OMS Query Language and QRL."

Creating Documents

QRS uses format files and QRL scripts to produce ASCII, HTML, FrameMaker, and Microsoft Word RTF (Rich Text Format) documents.

For details on format files and publishing tools, see Chapter 5, "Creating Documents."

Overview of QRS

In addition to the StP repository, QRS also accepts input from:

- User-supplied information, such as external variable values
- Values contained in QRL scripts

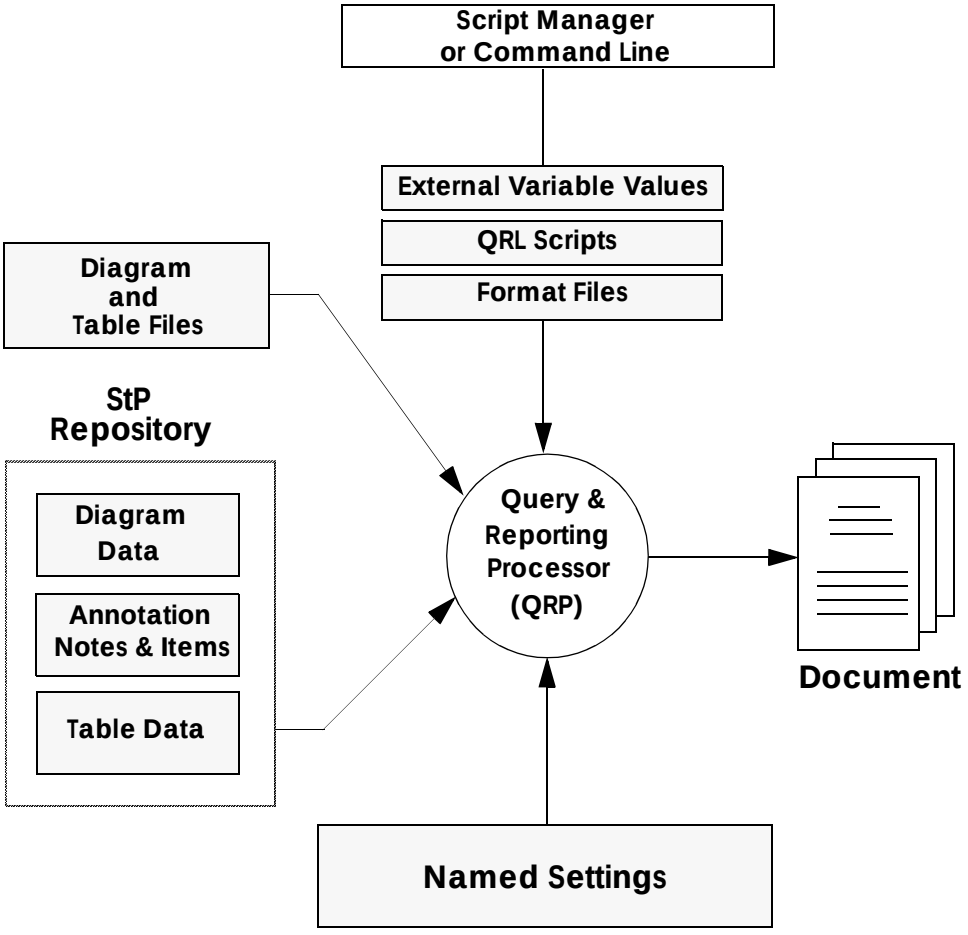
This information is processed by the Script Manager and formatted with settings specified in format files, **Print** command named settings files (for print options), and QRL scripts. The output is sent to one of the following:

- Standard output (usually your display)
- A file
- StP Repository Browser

Alternatively to using the Script Manager, you can also run the **qrp** command from the **Start Menu > Run** dialog box or from a command prompt in a shell window. For more information on using the qrp command, see "Running a Script with the qrp Command" on page 3-5.

Figure 1 shows an overview of how QRP uses scripts and information gathered from other parts of the StP environment to produce formatted documents.

Figure 1: Overview of the Query and Reporting System



2 The Script Manager

This chapter describes the main graphical interface for QRS, the Script Manager. The sections in this chapter are:

Using the Script Manager, you can:

- Run scripts to retrieve information from the repository, display that information in the Repository Browser, or include it in a formatted document
- Create, edit, delete, and copy scripts
- Provide values for external variables declared in scripts

There are several scripts supplied with StP that you can view, modify, and run through the Script Manager. You can also create your own scripts. For information on writing scripts, see Chapter 3, "The StP Query and Reporting Language." For information on including Object Management System (OMS) queries in scripts, see Chapter 4, "The OMS Query

Language and QRL." For information on using scripts to create documents, see Chapter 5, "Creating Documents."

This chapter describes how to access, create, modify, or run a script, or to assign values to its external variables using the graphical user interface of the Script Manager. However, it is not necessary to use the Script Manager to perform these tasks; a script can be created using any text editor anywhere on the system. Once a script is created, you run it from the Script Manager or from the command line in a shell window using the **qrp** command. The way in which you create, edit, and run scripts is a matter of personal preference.

For information on creating, modifying, and running scripts, see Chapter 3, "The StP Query and Reporting Language."

Starting the Script Manager

You start the Script Manager from the StP Desktop.

To start the Script Manager, choose **Start Script Manager** from the **Report** menu or use the **Start Script Manager** Toolbar button.

Alternatively, start the Script Manager from the command line by typing:

```
scriptman [-p <projdir>] [-s <system>]
```

where <projdir> is the name of the project directory and <system> is the name of the system. Items enclosed in brackets are optional.

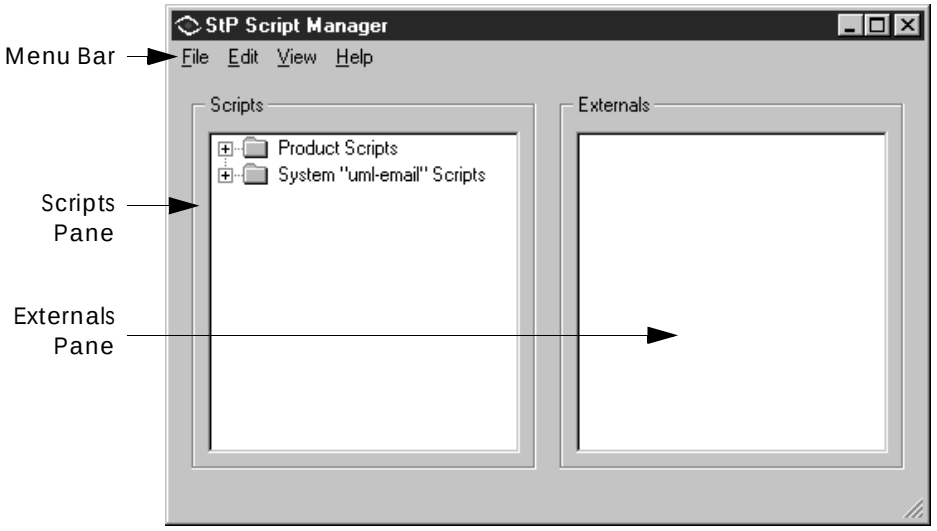
Using the Script Manager

This section describes the parts of the Script Manager.

The Script Manager Window

The Script Manager window, shown in Figure 1, is the primary interface of the QRS. This section discusses its various parts.

Figure 1: The Script Manager Window



Menu Bar

The Menu Bar contains menus. For information on these, see "Using the Script Manager Menus" on page 2-6. For information on displaying and using StP menus in general, see *Fundamentals of StP*.

Text Editor Window

Some commands in the Script Manager invoke a text editor window, such as **Edit External Value**.

The default text editor for the Script Manager is Notepad.

If you want to use a different text editor, you can specify it as the default for the following ToolInfo variables in your ToolInfo file:

- *scriptman_script_editor*
- *scriptman_external_editor*

For general ToolInfo variable information, see *StP Administration*.

For more information on ToolInfo files and a list of ToolInfo variables that apply to the Script Manager, see "ToolInfo Variables for the Script Manager" on page 2-22.

When a text editor window appears, the path and filename of the file are displayed. To use the text editor:

1. Type text.
2. Save and exit the file according to the rules of the text editor.

Scripts Pane

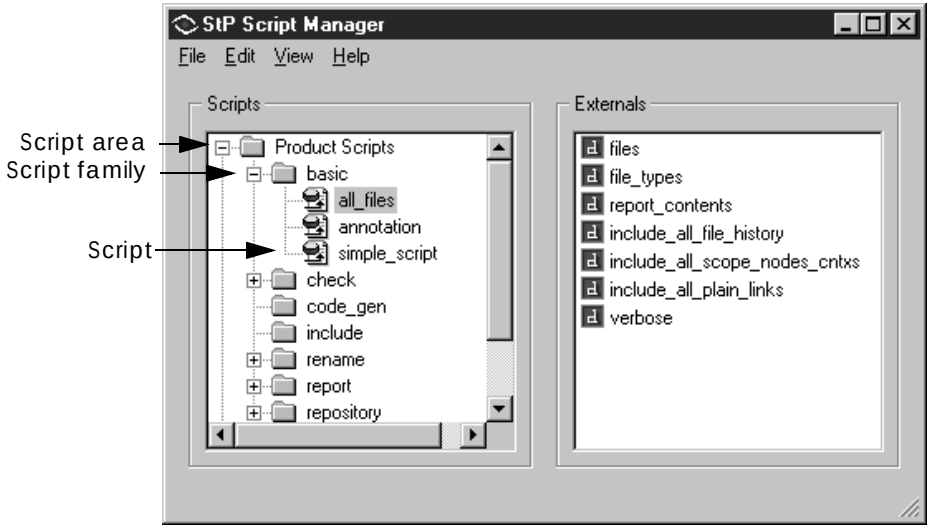
The **Scripts** pane provides a means for navigating the directory hierarchy where the QRL scripts reside.

This directory hierarchy contains general script areas, categories of scripts within script areas called script families, and scripts themselves, which reside in script families. Areas and families are directories (represented by folder icons); scripts are files (represented by file icons). Open areas, families, or files by single-clicking the "+" icon or double-clicking the folder icon or name.

For example, in Figure 1 on page 2-3, there are two script areas: *Product Scripts* and *System "uml_email" Scripts*, which are scripts in the current Project directory (in this case *uml_email*). In Figure 2 on page 2-5, the *Product Scripts* area is open, revealing script families. The script families shown in Figure 2 are particular to the current StP product. Some script families, such as *basic*, *check*, and *include*, are included in all StP products.

In Figure 2, the script family *basic* is open, displaying three scripts: *all_files*, *annotation*, and *simple_script*. The script *all_files* is selected, which causes the external variables declared in it to be displayed in the **Externals** pane.

Figure 2: Navigating the Scripts List



To display user scripts that are not located in the default StP location, use the scriptman_user_scripts_location ToolInfo variable. For more information, see "ToolInfo Variables for the Script Manager" on page 2-22.

Externals Pane

The **Externals** pane displays the external variables associated with a selected script. External variables are place holders whose values can be assigned at run time, either by using the Script Manager or the command line. The **Externals** pane is blank if the selected script has no external variables.

When a selected script has external variables, each variable is listed and preceded by an icon, as shown in Figure 2. An icon with the letter "d" indicates that the active value for the variable is the default; the letter "u" indicates a user-defined value. A round icon with an exclamation point "!" indicates that there is no default value and no value is assigned. For information on editing default values, see "Specifying Values for External Variables" on page 2-13.

Using the Script Manager Menus

The Script Manager provides five menus:

- **File**
- **Edit**
- **View**
- **Help**

Each menu lists commands and (if available) their corresponding access keys and shortcut keys.

File Menu

The **File** menu provides commands for running, terminating, or rescanning a script, and for exiting the Script Manager.

Table 1 describes the commands available from the **File** menu in the Script Manager.

Table 1: File Menu Commands

Command	Description	For Details, See
Run Script	Runs the selected script using your specifications for output file, format file, and whether to view the output file after the run.	"Running a Script from the Script Manager" on page 2-9
Terminate Script	Stops a script that is currently running.	"Terminating a Script" on page 2-13

Table 1: File Menu Commands (Continued)

Command	Description	For Details, See
Rescan	Updates the currently selected script in the **Scripts** list. For instance, if you are modifying a script by adding external variables, this command updates the **Externals** list to show the newly defined external variables. Also updates the **Scripts** list if outside scripts are copied into the scripts folder.	
Exit	Exits the Script Manager.	

Edit Menu

The **Edit** menu provides commands for editing, creating, copying, or deleting a script, or editing or deleting the value of an external variable.

Table 2 describes the commands available from the **Edit** menu in the Script Manager.

Table 2: Edit Menu Commands

Command	Description	For Details, See
Edit Script	Displays the selected script in a text editor window.	"Editing a Script" on page 2-22
Edit External Value	Allows you to enter a value for the selected external variable in a text editor window.	"Specifying Values for External Variables" on page 2-13
Create Script	Creates a copy of a script template in a text editor window. The new script is named according to your specifications.	"Creating a Script" on page 2-21

Table 2: Edit Menu Commands (Continued)

Command	Description	For Details, See
Copy Script	Copies the selected script to the specified name and location.	"Copying a Script" on page 2-19
Delete External Value	Restores the value of the selected external variable to the default.	"Deleting an External Variable's Value" on page 2-19
Delete Script	Deletes the selected script area, script family, or script.	"Deleting a Script" on page 2-22
Options	Provides commands for modifying the Script Manager and the message log.	*Fundamentals of StP*

View Menu

The **View** menu provides commands for displaying the Message Log and showing a script's path.

Table 3 describes the commands available from the **View** menu in the Script Manager.

Table 3: View Menu Commands

Command	Description	For Details, See
Show Message Log	Displays the Message Log.	*Fundamentals of StP*
Show Path	Displays the selected script's path in the message log.	

Help Menu

The **Help** menu provides commands for displaying online StP documentation and script help.

Table 4 describes the commands available from the **Help** menu in the Script Manager.

Table 4: Help Menu Commands

Command	Description
On-line Docs	Provides a list of online documentation in PDF (Portable Document Format). By clicking on a manual name, you navigate to the manual's table of contents.
About StP	Provides the current StP version number.
On Area	Displays help text about selection in a separate window. For information on writing customized help messages, see "Help" on page 3-83.
On Family	
On Script	
On External	

Scripts Shortcut Menu

The **Scripts** shortcut menu is accessible by pressing the right mouse button in the **Scripts** list area. This menu provides easy access to a subset of commands from the **File** and **Edit** menus.

Externals Shortcut Menu

Like the **Scripts** shortcut menu, the **Externals** shortcut menu is accessible by pressing the right mouse button in the **Externals** list area. It provides easy access to a subset of commands from the **Edit** menu.

Running a Script from the Script Manager

This section explains how to run a script from the Script Manager. It uses the script *all_files* as an example.

To run a script:

1. Start the Script Manager as instructed in "Starting the Script Manager" on page 2-2.
2. From the **Scripts** pane, find the script you want by clicking the appropriate area and family file icons to open them.

   The *all_files* script is in the *Product Script*s area, *basic* family.
3. Select the script you want to run.

   If the script includes externals, they appear in the **Externals** list.
4. (Optional) If the script includes externals, make any desired changes to their values.

   Procedures are described in "Specifying Values for External Variables" on page 2-13.
5. From the **File** menu, choose **Run Script**.
6. In the **Run Script** dialog box, make any desired changes to the default settings.

   For more information, see "The Run Script Dialog Box" on page 2-10.

   For FrameMaker, RTF (Microsoft Word), HTML, or non-default ASCII output, you must specify a format file, unless it is specified within the script. For information on format files, see "What Format Files Do" on page 5-10.
7. Click **OK** or **Apply**.

   An execution window appears and displays the script's messages, if any, while the script is running.

   To run multiple scripts, repeat the above steps for as many scripts as you wish to run.

The Run Script Dialog Box

When you choose **Run Script** from the **File** menu, the **Scripts** shortcut menu, or use the shortcut key, Control-R, the **Run Script** dialog box appears, as shown in Figure 3.

Figure 3: Run Script Dialog Box

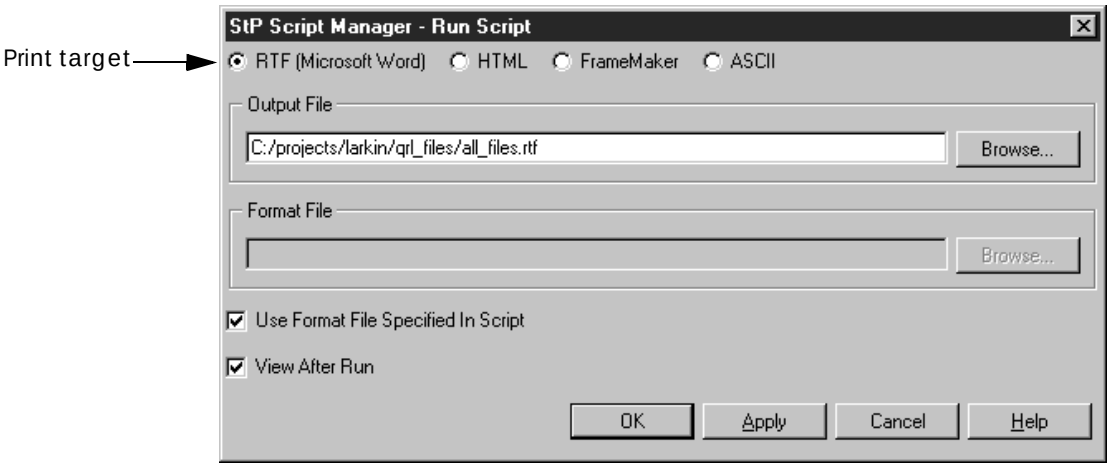


Table 5 shows the **Run Script** command properties.

Table 5: Run Script Command Options

Property	Description
Print Target	Sets a publishing target for the output of the script. Options are RTF (Microsoft Word), HTML, FrameMaker (MIF), or ASCII. The default is RTF (Microsoft Word).
Output File	Specifies a name and path for the output file. By default, the output file has the same name as the script with an extension that reflects the selected print target (.rtf for RTF, .html for HTML, .mif for FrameMaker, and .asc for ASCII). For information on specifying the path, see "Browsing Output Files," which follows.

Table 5: Run Script Command Options (Continued)

Property	Description
Format File	Specifies a format file to format the document produced by the script. If your print target is ASCII and you are using only system default formats, you do not need to specify a format file. In all other cases (for ASCII output that uses non-default formats, and all RTF, HTML, or FrameMaker output), you must specify a format file. For information on specifying format files, see "How qrp Locates Format Files" on page 5-8. For information on specifying the path, see "Browsing Format Files" on page 2-13.
Use Format File Specified in Script	When selected, this turns off the Format File option and specifies the format file to be used as the argument of the format function within the script. For information on the format function, see Table 2 on page 5-4.
View After Run	Displays the output document after the script is run. If the Print Target is HTML, the document is displayed in your default browser application (such as Microsoft Explorer or Netscape Navigator). If your target output is RTF, but you do not have Microsoft Word running when you run the script, the Script Manager starts MS Word after the document is generated. If your target output is FrameMaker, but you do not have FrameMaker running when you run the script, the Script Manager starts FrameMaker while the document is being generated. If your target output is ASCII, the document is displayed in the text editor associated with your Script Manager.

Browsing Output Files

To browse possible folders and files for the output file, in the **Run Script** dialog box, click **Browse** next to the **Output File** field.

In the **Choose File** dialog box, navigate to the output file and click **Open**. The default directory for HTML, FrameMaker, RTF, and ASCII is *<project> \ <system> \ qrl_files*.

Browsing Format Files

To browse possible folders and files for the format file, in the **Run Script** dialog box, click **Browse** next to the **Format File** field.

In the **Choose File** dialog box, navigate to the format file and click **Open**. For a list of the default folders, see Table 4 on page 5-8.

Terminating a Script

To terminate a script that is running:

1. From the **File** menu, choose **Terminate Script**.
2. In the **Terminate Script** dialog box, select the script you want to terminate.

   If no scripts are running, the dialog box is empty.
3. Click **OK** or **Apply**.

   The Message Log window appears with a message that the script is terminated.

Specifying Values for External Variables

A variable declared in a script for which you can assign values at run time is called an “external variable.” This section describes how to set values for external variables at runtime using the Script Manager. For information on how to set them on the command line, see “Running a Script with the qrp Command” on page 3-5.

For information on creating external variables, see "External Variables" on page 3-84.

When to Specify a Value at Runtime

When you declare external variables in a script, you can include default values, but this is not required. When you run the script, you must supply a value for an external variable if:

- No value is defined for the variable in the script
- You want to use a different value than the one defined in the script

Script Manager vs. Command Line Behavior

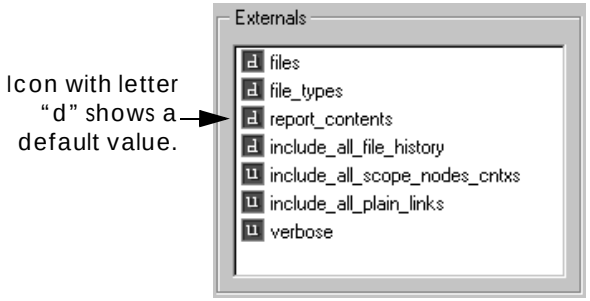
If you specify a non-default value for an external variable with the Script Manager, this value does not apply when you run the script from a command line. Conversely, if you specify an external variable value on the command line, this value does not apply when you run the script from the Script Manager.

When you specify a non-default value for an external variable with the Script Manager, it is retained until you change it. When you specify a non-default value for an external variable on the command line, it applies only during that execution of the command.

How Default Values Are Marked

In the **Externals** list of the Script Manager, external variables with default values have an icon with the letter "d," as shown in Figure 4. Variables with non-default values have icons with the letter "u." A round icon with an exclamation point "!" indicates that there is no default value and no value is assigned.

Figure 4: External Variable Values



Accessing Help on External Variables

Figure 5 shows you how to access Help on a specific external variable.

Figure 5: Accessing Help for External Variables

To access a description of an individual external variable:

1. Select an external variable.
2. From the Script Manager's **Help** menu, choose **On External**.

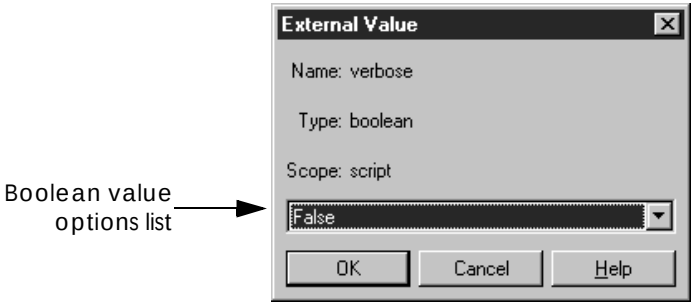
   Figure 6 shows the help dialog box for the external value selected in Figure 5.

Figure 6: Help Dialog Box.

Editing a Boolean External Value

Figure 7 shows the dialog box associated with a boolean variable. It shows the variable's name, type, scope, and current value.

Figure 7: External Value Dialog Box—Boolean



To edit a boolean external variable:

1. Select the script with the external variables you wish to modify.
2. Double-click the external variable in the **Externals** list.

   The **External Value** dialog box appears for the selected external variable.

3. In the **External Value** dialog box, choose the desired value from the Boolean value options list.
4. Click **OK** to apply the new value.

   The icon for the external value changes to a "u," indicating the variable has a non-default value.

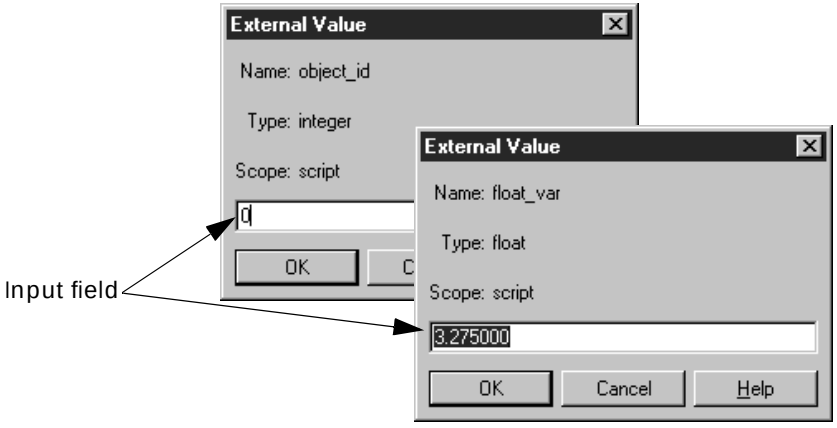
Alternatively, you can edit an external value in a text editor window by selecting the external variable and choosing **Edit > Edit External Value**.

Editing an Int or Float External Value

Figure 8 shows the dialog boxes associated with a variable of type int and float.

Both dialog boxes show the variable's name, type, scope, and current value. The value of an int or float external variable must be a number.

Figure 8: External Value Dialog Box—Int and Float



To edit an int or float external variable:

1. Select the script with the external variables you wish to modify.
2. Double-click the external variable in the **Externals** list.

   The **External Value** dialog box appears for the selected external variable.

3. In the **External Value** dialog box, type the desired value in the input field.
4. Click **OK** to apply the new value.

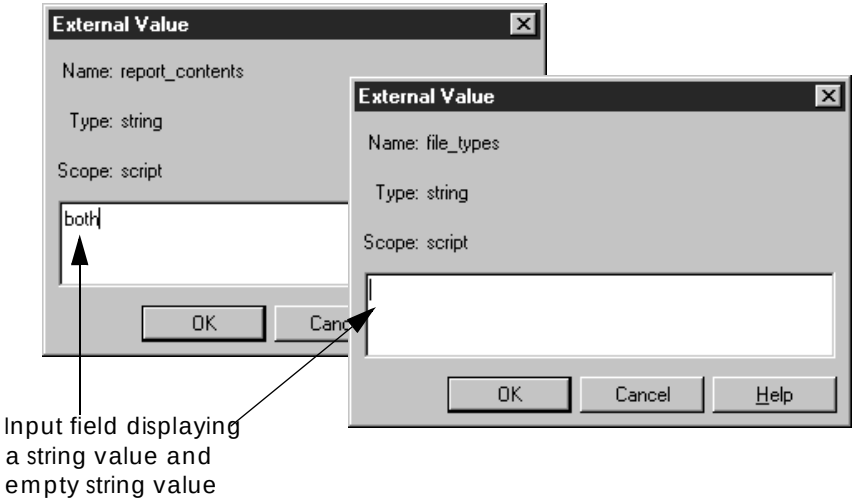
   The icon for the external value changes to a "u," indicating the variable has a non-default value.

Alternatively, you can edit an external value in a text editor window by selecting the external variable and choosing **Edit > Edit External Value**.

Editing a String External Value

When an external variable takes a string value, the **External Value** dialog box looks like the examples in Figure 9. The external variable *report_contents* has a string value; the external *file_types* has no value declared for it (an empty string).

Figure 9: External Value Dialog Box—String



To edit a string external variable:

1. Select the script with the external variables you wish to modify.
2. Double-click the external variable in the **Externals** list.

   The **External Value** dialog box appears for the selected external variable.

3. In the **External Value** dialog box, type the value into the input field of the dialog box.
4. Click **OK** to apply the new value.

   The icon for the external value changes to a "u," indicating the variable has a non-default value.

Alternatively, you can edit an external value in a text editor window by selecting the external variable and choosing **Edit > Edit External Value**.

Deleting an External Variable's Value

Deleting the value of an external variable causes the default value, as declared in the script, to be restored.

To delete an external variable's value:

1. Select the variable in the **Externals** list.
2. From the **Edit** menu, choose **Delete External Value**.
3. In the confirmation box, click **Yes**.

   The values in the **External Value** dialog box revert to the default values.

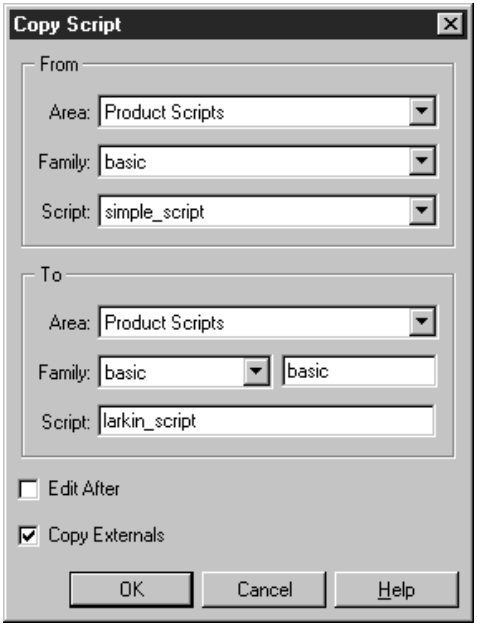
Copying a Script

You can use the Script Manager to copy a script from one location within the current system to another. You might, for example, want to copy a script from one script family into another.

To copy a script:

1. Select a script in the **Scripts** list.
2. From the **Edit** menu, choose **Copy Script**.

Figure 10: Create Script Dialog Box



3. In the **Copy Script** dialog box, verify the script you want to copy in the **From** group.
4. If necessary, display the **Area** and **Family** option lists in the **To** group to change the target location. Alternatively, type a new family name in the **Family** input field.
5. In the **Script** input field, type the destination script's file name.
6. Select **Edit After** if you want to edit the script after the copying is completed.
7. Select **Copy Externals** if you want to copy the original script's external variable values to the new script.
8. Click **OK**.

   The script is copied into the specified script area and family. If you elected to edit the script, a text editor window appears with the new script.

Creating a Script

Creating a script from the Script Manager is similar to copying a script, since the initial step copies a template script, *simple_script*, to a target location. The *simple_script* is included in the **Product Scripts** area.

To create a script, you must be familiar with QRL. For information about QRL, see Chapter 3, "The StP Query and Reporting Language."

To create a script:

1. From the **Edit** menu, choose **Create Script**.

2. In the **Create Script** dialog box, specify where you want to save the script when you finish creating it.

   If necessary, display the **Area** and **Family** option lists to change the target location. Alternatively, type a new family name in the **Family** input field.
3. To name the script, type the name of the script in the **Script** field.
4. If you want to edit the script, check **Edit After**.
5. Click **OK**.

   A text editor window appears, displaying a copy of *simple_script*. You can now create a script, using the *simple_script* template.

Editing a Script

To edit a script, you must be familiar with QRL. For information about QRL, see Chapter 3, "The StP Query and Reporting Language."

To edit a script from the Script Manager:

1. In the **Scripts** list, select the script you want to edit.
2. From the **Edit** menu, choose **Edit Script**.
3. Make the desired changes to the script, save the file, and exit the editor.

Deleting a Script

To delete an area, script family, or script:

1. In the **Scripts** pane, select the area, script family, or script you want to delete.

   Note: To delete a script area or family, the area or family icon must be open and empty.

2. From the **Edit** menu, choose **Delete <Area|Family|Script>**.
3. In the confirmation box, click **Yes**.

ToolInfo Variables for the Script Manager

ToolInfo files contain several variables that are relevant to the operation of the Script Manager. For more information on ToolInfo variables, see *StP Administration*.

ToolInfo variables are set in:

- *ToolInfo.W32NTX86*, which is the standard ToolInfo file installed in your StP product folder

- one or more user-specific ToolInfo files, which are used to set the StP features for a user's local environment

Script Manager ToolInfo variables are listed in Table 6.

Table 6: Script Manager ToolInfo Variables

Variable	Type	Default Value	Description
scriptman_ascii_view	String	notepad ${asciifile} &	Specifies the command to use to view ASCII output from the Script Manager.
scriptman_external_editor	String	notepad ${externalpath} &	Specifies the editor to be used by the Script Manager for editing external variables.
scriptman_product_ externals_location	String	Optional. No default value.	Provides the directory for product externals.
scriptman_script_editor	String	notepad ${scriptpath} &	Specifies the editor to be used by the Script Manager for editing scripts.
scriptman_script_process	String	qrp -p ${projdir} -s ${system} ${scriptpath} ${targetargs}	Specifies the process that executes QRL scripts from the Script Manager.
scriptman_system_ externals_location	String	Optional. No default value.	Provides the directory for system externals.
scriptman_user_ externals_location	String	Optional. No default value.	Provides the directory for user externals.
scriptman_user_scripts_ location	String	Optional. No default value.	Provides the location of user scripts.

Using Script Manager ToolInfo Variables

Certain Script Manager ToolInfo variables provide the capability for controlling which external variables are available to which users:

- All users can use the same set of externals for product scripts, system scripts, and user scripts

- Each user can use a different set of externals for product scripts, system scripts, and user scripts

Using One Set of External Values for One System

You can use Script Manager with the default ToolInfo settings.

The settings for product scripts are:

- Report source—*<product>_stp_file_path \ StP \ templates \ <product> \ qrl \ <family> \ <report>.qrl*
- Externals—*<project> \ <system> \ qrl_files \ product.ext \ <family> \ <report>.ext*

With these settings, all users who run the same product scripts for a system are using the same set of externals.

The settings for system scripts are:

- Report source—
  <project> \ <system> \ qrl_files \ scripts \ <family> \ <report>.qrl
- Externals—
  <project> \ <system> \ qrl_files \ scripts \ <family> \ <report>.ext

With these settings, all users who run the same system scripts for a system are using the same set of externals.

Adding User Scripts Location

If you want to add your own scripts to the Script Manager, set the ToolInfo variable *scriptman_user_scripts_location*. If you copy a product report to user scripts, the settings are:

- Report source—*<scriptman_user_scripts_location> \ <family> \ <report>.qrl*
- Externals—*<scriptman_user_scripts_location> \ <family> \ <report>.ext*

With these settings, all users who run the same user script for a system are using the same set of externals.

Using Separate External Values for One System

If you want each user to have individual Script Manager ToolInfo settings for product scripts (in their own ToolInfo files), set the ToolInfo variable *scriptman_product_externals_location*. In this case the settings are:

- Report source—<*product*>*_stp_file_path* \ *StP* \ *templates* \ <*product*> \ *qrl* \ <*family*> \ <*report*>*.qrl*
- Externals—<*scriptman_product_externals_location*> \ *product.ext* \ <*family*> \ <*report*>*.ext*

If you want each user to have individual Script Manager ToolInfo Settings for system scripts, set the ToolInfo variable *scriptman_system_externals_location*. In this case the settings are:

- Report source—<*project*> \ <*system*> \ *qrl_files* \ *scripts* \ <*family*> \ <*report*>*.qrl*
- Externals—<*scriptman_system_externals_location*> \ <*family*> \ <*report*>*.ext*

If you want each user to have individual Script Manager ToolInfo settings for user scripts, set the ToolInfo variable *scriptman_user_externals_location*. In this case the settings are:

- Report source—<*scriptman_user_externals_location*> \ <*family*> \ <*report*>*.qrl*
- Externals—<*scriptman_user_externals_location*> \ <*family*> \ <*report*>*.ext*

Summary

Table 7 is a quick reference guide to Script Manager tasks.

0

Table 7: Summary of Script Manager Tasks

To	Do
Run a script	Select a script and choose **Run Script** from the **File** menu or the **Scripts** pane shortcut menu, or use the shortcut (Control-R).
Update the currently selected script in the **Scripts** list	From the **File** menu, choose **Rescan**. For instance, if you are modifying a script by adding external variables, this command updates the **Externals** pane to show the newly defined external variables.
Exit the Script Manager	From the **File** menu, choose **Exit**.
Display the message log	From the **View** menu, choose **Show Message Log**.
View help on a script area, script family, an individual script, or an external	Select the item you want help on and choose **On Area**, **On Family**, **On Script**, or **On External** from the **Help** menu. If Help exists for the selected item, it appears in a separate window.
Display script path	Select a script or script family in the **Scripts** pane and choose **Show Path** from the **View** menu.
Display value of an external variable	Double-click an external variable in the **Externals** pane. This opens the **External Value** dialog box, showing the current value.

Table 7: Summary of Script Manager Tasks (Continued)

To	Do
Edit a script	Select a script in the **Scripts** pane and choose **Edit Script** from the **Edit** menu or the **Scripts** pane shortcut menu. A text editor window opens, showing information about the script.
Edit the value of an external variable	Select an external variable in the **Externals** pane, which brings up the dialog box for that variable. Or, choose **Edit External Value** from the **Edit** menu, which opens a text editor window, where you can adjust the value textually.
Create a script	From the **Edit** menu or the **Scripts** pane shortcut menu, choose **Create Script**. In the **Create Script** dialog box, specify the name and location for the new script, and click **OK**.
Copy a script	Select a script in the **Scripts** pane and choose **Copy Script** from the **Edit** menu or the **Scripts** pane shortcut menu. In the **Create Script** dialog box, specify a name and destination for the copied script, and click **OK**.
Delete an external value	Select the external variable whose value you want to delete in the **Externals** list and choose **Delete External Value** from the **Edit** menu or the **Externals** pane shortcut menu. This returns the value to the default.
Delete a script	Select the script you want to delete in the **Scripts** pane and choose **Delete Script** from the **Edit** menu or from the **Scripts** pane shortcut menu.

3 The StP Query and Reporting Language

This chapter introduces the StP Query and Reporting Language (StP / QRL). It describes:

This chapter describes the StP Query and Reporting Language (StP / QRL), a high-level, structured, C-like script language that includes familiar programming features such as types, parameterized function calls, and control constructs. QRL only reads repository information; it does not write to the repository.

While QRL is powerful enough for general purpose applications, it has specialized features for querying the repository and generating documents. You can use QRL to:

- Retrieve objects from the repository

- Generate documents
- Include text and graphics in documents

This chapter provides an overview of QRL. Chapter 4, "The OMS Query Language and QRL," describes how to use OMS queries in a QRL script to extract objects from the repository. Using QRL scripts to generate documents is covered in Chapter 5, "Creating Documents."

For information about the Script Manager, graphical interface to the QRS, see Chapter 2, "The Script Manager."

QRL Basics

You create QRL scripts by modifying ones that already exist on your system or by creating entirely new ones. When you have finished your script, you run it using the Script Manager or on the command line with the **qrp** command.

Keep in mind that, unlike a C program, which is compiled, QRL scripts are interpreted.

Using Existing Scripts

There are several QRL scripts distributed with StP. Depending on which StP products you use, these scripts may perform such tasks as:

- Defining the behavior of diagram and table editors
- Aiding in passing messages between StP and third-party applications
- Providing checking functions for objects in the repository
- Generating code and / or schema
- Generating diagram information reports

You can access these scripts through the Script Manager on the StP Desktop or through the command line. Accessing them through the Script Manager is described in "Scripts Pane" on page 2-4. To see which scripts are available, examine the contents of the following directories:

- *<product>_stp_file_path\templates\ct\qrl*—for scripts provided with StP Core
- *<product>_stp_file_path\templates\<product>\qrl*—for scripts provided with your StP product

Subdirectories under these directories are the script families viewable in the Script Manager; the files in the subdirectories named with a *.qrl* extension are the QRL scripts.

Creating New Scripts

You can use QRL to create your own scripts. A QRL script is built from a number of basic programming elements, such as types, declarations, and control constructs. These elements are described in the following sections.

Basic Components of a QRL Script

Figure 1 illustrates a simple example of QRL.

Figure 1: Sample Script: *script_example.qrl*

```
// simple script example
void
main ()
{
    print_line("Hello, world.");
}
```

This simple script introduces a number of important elements in QRL:

- The first line of the script is a comment:

  ```
  // simple script example
  ```

  A comment provides information about a script, can appear anywhere in the script, and must begin with two slashes (/ /) at the beginning of the line. All text that follows the slashes on a single line is ignored when the script is run.

QRL also supports C-style comments, where all text between `/*` and `*/` is ignored:

```
/* this is a comment */
             /* so is this */
/*
and this
*/
```

You cannot have executable statements on a line that starts with a comment.

- The keyword “void” declares the type of value returned by the main function.

  All functions must be declared with a return type. In this example, main does not return a value and therefore uses a special return type, void.

- The “main()” function indicates where the script begins executing.

  A QRL script must contain a “main” function.

- The main function is followed by an empty open and close parentheses.

  All function declarations in QRL must include open and close parentheses; this is where the function’s parameters (if any) are declared. The open and close parentheses with nothing between them indicate that the main function does not have any parameters.

- The print_line function is a built-in function that prints its arguments and then a new line. The argument here is

  `"Hello, world."`

  A built-in function is a function that is included as a part of QRL; QRL has many such built-in functions.

- The quotation marks around `"Hello, world."` indicate that it is a literal string.

- The print_line statement is an example of a simple statement. The semi-colon after the closing parenthesis of the print_line statement is required.

Running a Script with the qrp Command

In addition to running a script from the Script Manager, you can run a script from the command line by using the **qrp** command. To run a script from the command line, type the following:

```
qrp <script_name>
```

Running the script displays this output in the shell window. For Figure 1, the output is:

```
Hello, world.
```

qrp Command Syntax

The syntax for the **qrp** command is:

```
qrp <script_name> | -C <qrl_command> [-p <projdir>]
    [-s <system>] [-t <target>] [-f <format_file>]
    [-o <output_file>] [-I <include_dir>]...
    [-x <ext_var_name> <ext_var_value>]...
    [-trace <arguments>] [-trace_file <file>]
    [-version]
```

Table 1 describes the **qrp** command options, except for the **trace** and **trace_file** options, which are described in "QRL Debugging Features" on page 3-91.

Table 1: qrp Command Options

Option	Argument	Description
	<script_name>	Specifies the QRL script to be used in generating the output. (This argument does not use an option flag.)
-C	<qrl_command>	Specifies a qrl command, such as: `print(time_to_string(time_now(), NULL));`

Table 1: qrp Command Options (Continued)

Option	Argument	Description
-p	<projdir>	Specifies the current project directory. This directory overrides the default directory specified by the *projdir* ToolInfo variable.
-s	<system>	Specifies the current system name. This name overrides the default system specified by the *system* ToolInfo variable.
-t	<target>	Specifies the target output system. Values of <target> can be “ascii” (for ASCII), “html” (for HTML), “mif” (for FrameMaker), or “rtf” (for RTF). You do not need to specify ascii, since it is the default.
-f	<format_file>	Specifies the format file to be applied to the output. For complete information on specifying format files, see “How qrp Locates Format Files” on page 5-8.
-o	<output_file>	Specifies the full path and file to which you want to write the script’s output (stdout is the default).
-I	<include_dir>	Specifies include paths other than those defined by the *stp_file_path* ToolInfo variable.
-x	<ext_var_name> <ext_var_value>	Specifies the name of an external variable used in a script and a value for that variable.
-version		Displays version information for the StP Core and, if appropriate, product components. Does not perform the normal action of the program.

Variables, Data Types, and Operators

This section describes how to use variables, data types, and operators in QRL.

Variables

Variables are symbolic names that store values. Variables must be of a particular type, such as int or string, which determines the values it can have and what operations can be performed on it. For example, you cannot assign the string “abc” to a variable of the type int.

In a QRL script:

- A variable must be declared before it is used.
- A variable must be declared to be a specific type.
- Variable declarations can occur anywhere in the script. There is no declare block.
- Variables declared within a function are local to that function; variables outside of functions are globally scoped.

  For more information on variable scoping, see “Variable Scoping” on page 3-31.
- Referencing an undeclared variable generates an error.

Declaring Variables

Variable declarations use this syntax:

```
<type> <variable> [= <initval>] [,<variable>...];
```

For example:

```
string s;
int int_var;
```

You can declare a list of variables, separated by a comma, in a single statement:

```
string s, n;
```

You can also initialize a variable or a series of variables in a declaration statement. For example, the following statement declares the variables "var1" and "var2" and provides initial values for them:

```
int var1 = 10, var2 = 15;
```

Variable Names

A variable name can be up to 255 characters long and can use any letter, including the underscore, for the first character. For example, _top, a1, and z9 are all legal variable names.

Reserved Words

Like most programming languages, QRL has words that have a special meaning in the language. The following words are reserved, and therefore cannot be used as variable names:

all type names	external	return
break	False	script_help
const	for	struct
continue	for_each_in_select	system_external
else	if	True
enum	include	while
external_help	NULL	

QRL Data Types

The declaration of a variable specifies its type, which determines how that variable is used in the script.

QRL supports the following categories of data types:

- Primitive types

  The primitive types are int, float, string, and boolean.
- PDM (Persistent Data Model) types

PDM types correspond to abstract data types defined by the Object Management System (OMS) for modeling persistent data maintained in the repository. For a list of PDM types, see "PDM Types" on page 3-13. For a complete description of PDM types and their attributes, see *Object Management System*.

- User-defined types

  The user-defined types are enumeration types and structures. For more information, see "User-Defined Types" on page 3-34.

- QRL abstract types

  The QRL abstract types are list, set, and graph. Both lists and sets are collections of elements. For more information, see "QRL Abstract Types" on page 3-39.

- QRL printing types

  For information on these, see "Enumeration Types for Print Options" on page 5-50.

These data types are described in detail below.

Primitive Types

Conceptually, primitive types correspond to the basic data types in the C language. They are int, string, float, and boolean. You can declare more than one variable to be of a particular type in a single statement, and you can also initialize variables in a declaration statement. For example:

```
int int_var1, int_var2;
string s = "This is the value of s.";
float float_var = 603.5;
boolean is_key = True;
```

Table 2 lists the primitive types, their binary and unary operations, and the values that can be assigned to them.

Table 2: Primitive Types

Type	Binary Legal Operations	Unary Legal Operations	Values
int	+ - * / % == != < > <= >= && | |	!-	Any whole number
float	+ - * / == != < > <= >=	-	Any number with a fractional part
string	+ != == < <= > >=		Any string of characters
boolean	== != && | |	!	True, False

Converting Types

QRL includes built-in functions that enable you to convert a variable from one primitive data type to another. For example, the script in Figure 2 converts a floating point value to an integer value and prints the results.

Figure 2: Sample Script: Converting Types

```
void
main()
{
   float a = 3.275;
   int b;
   // convert to int type
   b = to_int(a);
   print_line("The floating point value of 'a' is " + a);
   print_line("The integer value of 'a' is " + b);
}
```

The output of this script is:

```
The floating point value of 'a' is 3.275000
The integer value of 'a' is 3
```

The to_int function is described in Table 3.

A plus sign (+) in the print_line statements is used to concatenate a character string and the string representation of a variable's value. Unlike the way it is used in an arithmetic statement, the + operator is not used to add the literal strings and the variables together.

Table 3 lists the type conversion functions available in QRL.

Table 3: Type Conversion Functions

Function	Return Type	Description
to_string (any primitive type <value>)	string	Returns a string representation of <value>.
to_int (float, string, or boolean <value>)	int	Converts the float, string, or boolean <value> to an integer.
to_boolean (int or string <value>)	boolean	Converts the int or string <value> to a boolean <value>.
to_float (int or string <value>)	float	Converts the int or string <value> to a float <value>.

Table 3 introduces the convention used for representing function syntax in this document:

```
function_name([<type param_1>, <type param_2>...])
```

where each <type param> pair indicates an argument that must be supplied to the function. When you declare and call the function in a script, you must supply arguments that match the order and type of arguments shown in the function's syntax statement. For more information on declaring and calling functions, see "How Functions Work in QRL Scripts" on page 3-28.

The Return Type column in Table 3 lists the function's return type.

Character Strings

You can include special formatting characters within string literals. For example, including a backslash followed by the letter n in a string literal inserts a new line at that point in the string:

```
print_line("Hello, \n world.");
```

The output of this line is:

```
Hello,
world.
```

To include a tab within a string literal, use \t:

```
print_line("Hello, \t world.").
```

The output of this line is:

```
Hello,    world.
```

You can use the backslash character to "escape" the meaning of special characters and character sequences in QRL. For example, to include the literal string \t in a character string, precede \t with another backslash:

```
print_line("Include \\t to set a tab within a string.");
```

The output of this line is:

```
Include \t to set a tab within a string.
```

Similarly, to include quotation marks within the string itself, type a backslash before the quotation mark:

```
print_line("The name\"QRL\" stands for Query and " +
             "Reporting Language.");
```

The output of this line is:

```
The name "QRL" stands for Query and Reporting Language.
```

Note that the above print_line statement is broken and concatenated in the middle of a text string. This illustrates another use of the plus sign (+) in a print_line statement, namely, to concatenate short strings into longer ones in the output. This helps to avoid line wrapping and makes scripts easier to read.

PDM Types

The Object Management System (OMS) defines a set of abstract types used to model persistent data maintained in the repository. QRL has a set of corresponding PDM data types that allow access to repository objects.

The PDM types are:

- cntx
- cntx_ref
- file
- file_history
- file_lock
- item
- link
- link_ref
- node
- node_ref
- note
- viewpoint

You declare a variable for a PDM type just as you declare any variable:

```
node node_var;
file file_var;
```

For all the PDM types, the only legal operators are == and `!=`.

For more information about PDM types and the OMS, see the *Object Management System* manual. For more information on using PDM types in scripts, see Chapter 4, "The OMS Query Language and QRL."

Attributes of PDM types can be evaluated but not changed (that is, an attribute cannot be the target of an assignment).

Constants

A constant is an object of any type whose value cannot be changed. When you define a constant, it cannot be used as the target of an assignment or

as an argument to the function that changes its value. Constants must be initialized at declaration time.

Syntax:

```
const <type> <variable> = <initialization expression>
```

For example:

```
const boolean REQUIRED = True;
const int INT_CONST = 400;
```

The example in "Using the if-else Statement" on page 3-26, and the example in "String Manipulation Functions" on page 3-58 are scripts that use constants.

The NULL Constant

The keyword NULL refers to a constant NULL value of any type. NULL can be both assigned and compared to a variable of any type.

All other operations with NULL are illegal.

QRL Operators

The StP Query and Reporting Language uses common assignment, arithmetic, and logical operators.

Arithmetic Operators

QRL uses the same arithmetic operators as many other programming languages. Table 4 lists the arithmetic operators available in QRL; these operators are called binary operators since they operate on two values or arguments.

Table 4: Arithmetic Operators

Operator	Name/Definition
+	addition

Table 4: Arithmetic Operators (Continued)

Operator	Name/Definition
-	subtraction
*	multiplication
/	division
%	remainder

The remainder operator is similar to the division operator, in that, given the expression, `x % y = z`, it divides x by y. The difference is that the value of z is the remainder of the division. This, and other arithmetic operators, are illustrated in the script in Figure 3.

Figure 3: Sample Script: Arithmetic Operators

```
void
main()
{
    int a = 80;
    int b = 7;
    int c = 20;
    int d = 5;
    int result;
    // subtraction
    result = a - b;
    print_line("a - b = " + result);
    // multiplication
    result = b * d;
    print_line("b * d = " + result);
    // division
    result = a / c;
    print_line("a / c = " + result);
    // precedence
    result = a + b * c;
    print_line("a + b * c = " + result);
    // remainder
    result = a % b;
    print_line("a % b = " + result);
    // calculate the result within the print_line
    // statement itself
```

```
    print_line("a * b + c * d = " + (a * b + c * d));
}
```

The output of this script is:

```
a - b = 73
b * d = 35
a / c = 4
a + b * c = 220
a % b = 3
a * b + c * d = 660
```

In Figure 3, notice that the order in which arithmetic operations are performed is significant. QRL follows the rules of associativity and precedence common to most computer languages: evaluation usually proceeds from left to right, but multiplication and division are performed before addition and subtraction. For instance, in the calculation 3 + 4 * 2, 4 is multiplied by 2 before 3 is added. The total is 11.

If you want to alter the order of operations, use parentheses. For instance, (3 + 4) * 2. In this calculation, 3 is added to 4 before the figure is multiplied by 2. The total is 14.

Assignment Operators

An assignment operator assigns a value to a data object. The basic assignment operator in QRL is Assign (=). QRL also includes some of the standard C shortcut assignment operators. Table 5 lists the assignment operators available in QRL and the equivalent expressions for the shortcuts.

Table 5: Assignment Operators

Operator	Name	Definition
=	assign	
+=	add assign	a += b is the same as a = a + b
-=	subtract assign	a -= b is the same as a = a - b
*=	multiply assign	a *= b is the same as a = a * b

Table 5: Assignment Operators (Continued)

Operator	Name	Definition
/=	divide assign	a /= b is the same as a = a / b
%=	remainder assign	a %= b is the same as a = a % b

Relational and Logical Operators

QRL uses common relational and logical operators. Relational operators are used to test specific conditions, such as “less than” or “equal to.” The answer to a relational test is True or False. For example:

```
if (game_is_over == True)
   print("The game is over.");
```

Alternatively, this could be expressed as:

```
if (game_is_over)
   print("The game is over.");
```

Logical operators, AND (&&) and OR (||), are used to perform compound relational tests, such as:

```
//if i is greater than 1 AND less than 10, execute
//print statement
if ( i > 1 && i < 10 )
   print(i + "is a number between 1 and 10");
```

Or:

```
//if index is less than 0 OR greater than 10, execute
//print statement
if (index < 0 || index > 10)
   print("Index is out of range");
```

The relational and logical operators supported by QRL are listed in Table 6. With the exception of “not” (!), these operators are binary, meaning that they operate on two values or arguments.

Table 6: Relational and Logical Operators

Operator	Name/Definition
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	is equal to
&&	and (binary)
||	or (binary)
!=	not equal (binary)
!	not (unary)

Relational and logical operators have the same rules of precedence as the C language:

- Relational operators all have the same precedence.
- The operators == and != have the same precedence.
- The logical operator && (and) has a higher precedence than || (or).

Unary Operators

QRL includes four unary operators that increase or decrease the value of an expression by one. Table 7 lists these operators and describes their function.

Table 7: Incrementing & Decrementing Unary Operators

Operator	Name	Definition
++<expr>	Preincrement operator	Increases the value of <expr> by 1 before returning the value.
--<expr>	Predecrement operator	Decreases the value of <expr> by 1 before returning the value.
<expr>++	Postincrement operator	Increases the value of <expr> by 1, but returns the original value.
<expr>--	Postdecrement operator	Decreases the value of <expr> by 1, but returns the original value.

The Postincrement and Postdecrement operators are useful when the expression they are applied to consists of an object that can store the new value. This is illustrated in Figure 4.

Figure 4: Sample Script: Using a Postincrement Operator

```
void
main()
{
    int i = 0;
    int upper_bound = 10;

    while(i < upper_bound)
    {
        // Print the original value and
        // increment the variable
        print(i++);
    }
}
```

The output from this script is:

```
0123456789
```

Intertype Operations

You can perform the following intertype operations in QRL:

- string + <any other type>

  Converts the non-string to a string and concatenates it to the original string. For example, the statement:

  ```
  print("abc" + int_var);
  ```

  converts the integer variable int_var to a string and concatenates it to the string “abc”.
- int <any operation> float

  converts the int to a float and performs any legal float operation.

All other intertype operations are illegal. QRL includes built-in functions to perform explicit type conversion; for more information, see “Converting Types” on page 3-10.

The script in Figure 5 illustrates intertype operations. It converts a float, flt_var, to a string and concatenates it to another string, string_var. Then it converts an integer, int_var, to a float, and divides the float flt_var by int_var.

Figure 5: Sample Script: Type Conversion

```
void
main()
{
   float y, flt_var = 8.9235;
   int int_var = 788;
   string x, string_var = "this is a string:";
   // convert flt_var to string, concatenate to string_var
   x = string_var + flt_var;
   // convert int_var to float, divide flt_var by int_var
   y = flt_var / int_var;
   print_line("The value of x is \"" + x + "\"");
   print_line("The value of y is " + y);
}
```

The output from this script is:

```
The value of x is "this is a string:8.923500"
The value of y is 0.011324
```

Control Flow

QRL uses statements and constructs to direct the flow of the script, including loop constructs and an if-else statement.

Statements and Blocks

An expression followed by a semi-colon (;) is a simple statement. For example, the expression `a = b;` is a statement.

Statements grouped together within curly braces { } constitute a block or compound statement.

Loop Constructs

Using loop constructs enables a script to execute a set of statements repetitively. QRL uses “for,” “while,” and “do-while” for program looping, and “break” and “continue” to affect loop processing when certain conditions occur.

Using a for Loop

A for loop typically starts with an initialized loop index, and increments or decrements the index with each iteration through the loop. The loop ends when a condition is no longer met. As long as the condition is met, the statement or statements within the loop are executed.

The general form of the for loop is:

```
for(initial-statement(s); condition; iteration-statement(s))
{
    loop-body
}
```

The example in Figure 6 illustrates a for loop.

Figure 6: Sample Script: A for Loop

```
void
main()
{
   int ix;
   for (ix = 1; ix <= 5; ix = ix + 1)
   {
      print_line(ix);
   }
}
```

Briefly, this is how the above script is performed:

1. The initial-statement (`ix = 1;`) is executed.
2. The condition (`ix <= 5;`) is checked.

   If the condition is false (ix > 5), the script skips the loop-body and terminates.
3. If the condition is true, the loop-body (`print_line(ix);`) is executed.
4. The iteration-statement (`ix = ix + 1`) is executed.
5. Go back to step 2 (checking the condition).

The for loop is executed five times. The output of the script is:

```
1
2
3
4
5
```

Using a while Loop

A while loop starts with an evaluation of an initial condition. The loop ends when a condition is no longer met. As long as the condition is met, the statement or statements within the loop are executed.

The general form of the while loop is:

```
while(condition)
{
```

```
    loop-body
}
```

The example in Figure 7 shows the previous for loop script (Figure 6 on page 3-22) modified to use a while loop instead.

Figure 7: Sample Script: A while Loop

```
void
main()
{
    int ix = 1;
    while (ix <= 5)
    {
        print_line(ix);
        ix = ix + 1;
    }
}
```

This is how the above script is performed:

1. The initial-statement (`ix = 1;`) is executed.
2. The condition (`ix <= 5`) is checked to see if it is true.
3. If the condition is false (ix > 5), the script skips the loop-body and terminates.
4. If the condition is true, the statements in the loop-body (`print_line(ix);` and `ix = ix + 1;`) are executed.
5. Go back to step 2.

The while loop is executed five times and produces the same result as the for loop in the previous example. In general, though you can produce the same results using a for loop that you can using a while loop, a for loop is used when:

- An initialization condition exists
- Per-loop processing (such as incrementing a counter) occurs at the end of the loop body

Using a do-while Loop

A do-while loop is similar to a while loop, except that the evaluation test is at the bottom of the loop. The loop ends when a condition is no longer met. As long as the condition is met, the statement or statements within the loop are executed.

The general form of the do-while loop is:

```
do
{
    loop-body
}
while(condition);
```

The example in Figure 8 substitutes a do-while loop for the while loop of the previous script (Figure 7 on page 3-23).

Figure 8: Sample Script: A do-while Loop

```
void
main()
{
    int ix = 1;
    do
    {
        print_line(ix);
        ix = ix + 1;
    }
    while (ix <= 5);
}
```

This is how the above script is performed:

1. The initial-statement (`ix = 1;`) is executed.
2. The statements in the loop-body (`print_line(ix);` and `ix = ix + 1;`) are executed.
3. The condition (ix <= 5) is checked to see if it is true.
4. If the condition is false (ix > 5), the script terminates.
5. If the condition is true, go back to step 2.

The do-while loop is executed five times and produces the same result as the while loop in the previous example. In general, though you can produce the same results using a while loop that you can using a do-while loop, use a do-while loop when you want to guarantee at least one execution of the loop-body.

Using the break Statement

You can use a break statement in a do-while, for, or while loop. When the break statement is executed, it causes the loop to be exited immediately. If the loop in which the break statement occurs is nested within an outer loop, the inner loop is exited and execution resumes in the outer loop.

In the script in Figure 9, the while loop continues executing until the specified termination condition (i == 0) occurs.

Figure 9: Sample Script: A break Statement

```
void
main()
{
   int i = 10;
   print_line("Countdown:");
   // while 1 (that is, while TRUE), continue executing
   // until termination condition occurs
   while(1)
   {
      print(i + "...");
      i = i - 1;
      if(i == 0)
      {
             print_line("\nBlastoff!");
             break;
      }
      }
}
```

The output of this script is:

```
Countdown:
10...9...8...7...6...5...4...3...2...1...
Blastoff!
```

You can also achieve the same results by using a for loop, or by using the statement `while(i != 0)` instead of placing the termination condition within the loop.

Using the continue Statement

Like break, the continue statement can only be used within a for or while loop. The continue statement causes subsequent statements within a loop body to be skipped, and returns control to the top of the loop where execution of the loop continues as normal. If the loop is a for loop, the iteration step or statement is executed.

For an example of a script that uses a continue statement, see Figure 15 on page 3-47.

Using the if-else Statement

QRL includes an if-else statement, which is used to make decisions. The syntax of this statement is:

```
if (condition)
   statement1
else
   statement2
```

If the condition specified in the `if(condition)` statement is met, then `statement1` is executed. If not, then `statement2` is executed. Note that the `else` part of the statement is optional; it is possible to use an if statement without a corresponding else statement.

QRL also supports the compound statement else if:

```
if (condition1)
   statement1
else if (condition2)
   statement2
else
   statement3
```

If `condition1` is true, `statement1` is executed. Otherwise, if `condition2` is true, `statement2` is executed. Otherwise, `statement3` is executed.

The script in Figure 10 determines whether the members of a series of integers are odd or even and prints the results. The script includes a constant and a for loop.

Figure 10: Sample Script: Using if-else Statements

```
// Declare a constant N with the value 10
const int N = 10;
void
main()
{
   int i, result1, result2;
   for (i = 1; i < N; i = i + 1)
   {
      // Divide i by 2
      result1 = i / 2;
      // Subtract result1 * 2 from i
      result2 = i - result1 * 2;
      // Test to see if result2 != 0; if so print that
      // i is odd
      if(result2 != 0)
            print_line(i + " is an odd number");
      // Else print that i is an even number
      else
            print_line(i + " is an even number");
   }
}
```

The output of this script is:

```
1 is an odd number
2 is an even number
3 is an odd number
4 is an even number
5 is an odd number
6 is an even number
7 is an odd number
8 is an even number
9 is an odd number
```

In this script:

- The if statement is evaluated first.
- If it is true, then the associated print_line statement is executed.

- If it is false, the else statement and its associated print_line statement are executed.

How Functions Work in QRL Scripts

One of the fundamental elements in all QRL scripts is the function. A function encapsulates a particular computation; functions provide a way of packaging code and giving that package a name that can be referred to by other functions.

QRL supports both user-defined and built-in or pre-defined functions (the print_line function used in the previous examples is a built-in function). QRL built-in functions include:

- OMS interface functions
- Printing functions
- Browse functions
- String manipulation functions
- Primitive type conversion functions
- System functions
- Enumeration type functions
- Abstract type functions

The built-in functions are described in the remainder of this chapter, in Chapter 4, “The OMS Query Language and QRL,” and in Chapter 5, “Creating Documents.”

A script must include a main function, which is where execution of the script starts. The main function is declared to return the type void, or, if the script has an exit status, int. The return value of main is used as the exit status of the script.

A function has four parts:

- A return type
- The function name
- A parameter profile

- The function body

The parameter profile, which is enclosed within parentheses, contains a comma-separated list of zero or more parameters.

The function body is enclosed within curly braces; the open and closed curly braces indicate where a function begins and ends, respectively. The function body contains a sequence of program statements.

Declaring Functions

The syntax for function declaration is:

```
<return_type>
<function>([<param_declaration>] [, <param_declaration>]...)
{
   <statements>
   [return [<expression>];]
}
```

A return statement is required for all functions, except for functions declared as void.

The <param_declaration> consists of a <type> <parameter> pair. Multiple parameter declarations are separated by commas. For example:

```
int func1(string <var1>,int <var2>)
```

where func1 is a function returning the type int. The func1 function has two parameters: <var1> and <var2>, which are of the types string and int, respectively. Just as in C, there is no distinction between procedures and functions. If you want "procedure-like" subroutine invocation (that is, no meaningful value is returned), declare the return type as void:

```
void <function_name>(<parameter_declaration(s)>)
```

A function's return type can be any data type supported by QRL, including user-defined types.

Calling Functions

A function call has the following format:

```
<function>([<argument>] [, <argument>...]);
```

Each item in the argument list corresponds positionally to a parameter in the function definition. Any type mismatch between an argument and a formal parameter (that is, a parameter specified in the function's definition) generates an error during the function call. In other words, if there is a parameter mismatch, the function is not called.

For example, given the function func1:

```
int func1(string <var1>, int <var2>)
```

To call `func1` from another function, use:

```
func1("this is the string value for var1", 5);
```

where `"this is the string value for var1"` is the argument for <var1> and 5 is the argument for <var2>.

Note: QRL abstract types (lists, sets, and graphs) and structures are passed by reference, while all other types are passed by value. That is, passing a variable of type list, set, or graph to a function can result in change to a variable that is visible to the calling function. For more discussion of this subject and an example, see "Passing list, set, and graph Values" on page 3-89.

QRL functions can be called recursively.

The example in Figure 11 includes a main function and a user-defined function, change_it. It illustrates how arguments are passed by value. In the script, the subfunction change_it takes the value of i from main, assigns it to k, adds 21 to k, and returns the value of k to main. The value of i does not change.

Figure 11: Sample Script: Using Built-In and User-Defined Functions

```
void
main()
{
   int i = 3, j;
   // Print value of i before change_it
   print_line("i = " + i + " before change_it");
   // Call change_it function with argument i
   j = change_it(i);
   // Print values of i and j after change_it
   print_line("i = " + i + ","+ " j = " + j +
```

```
                " after change_it");
}
int
change_it(int k)
{
    // Print value of k; k has value of i at this point
    print_line("k = " + k);
    // Change value of k by adding 21 to it
    k = k + 21;
    // Print new value of k
    print_line("k = " + k);
    // Return value of k to main
    return k;
}
```

The output from this script is:

```
i = 3 before change_it
k = 3
k = 24
i = 3, j = 24 after change_it
```

Variable Scoping

Where a variable is declared in relation to functions determines the variable's scope:

- Local variable—A variable declared within a function

  A local variable is limited in scope to that function.
- Global variable—A variable declared outside of functions

  A global variable can be used by any function to which it is visible. For example, if a global variable is declared in an included file, it can be used by any script that includes that file. (See "Using the #include Command" on page 3-32 for more information.)

Standard programming language scoping rules apply:

- A local variable can only be referenced within the scope in which it is declared.
- A local variable overrides (hides) a global variable with the same name.

- A global variable can be referenced anywhere in the script as long as it has been declared in the file or in an included file.

Note: Local variables should not be declared within a loop. For a more detailed explanation, see "Declaring Local Variables" on page 3-88.

Using the #include Command

The **#include** command allows you to include QRL definitions and declarations contained in other files in a script. Included files typically contain information that can be used by other scripts, such as:

- Structure definitions
- Constants
- Enumeration types
- Common functions
- Global variables

The effect of using **#include** is to provide visibility to all QRL declarations in the included file. For this reason, it is important to avoid naming collisions between functions defined in the script in which the command is executed and in the included files. If there is a naming collision, the Script Manager or **qrp** reports an error when you attempt to run the script.

The syntax for including a file is:

```
#include "<filename>"
```

For example:

```
#include "/qrl/include/file.inc"
```

The **#include** command lexically replaces the entire include statement with the contents of <filename>. The search path for <filename> is:

- If <filename> is a full pathname, QRS searches for a file with that name
- Otherwise, QRS searches the file relative to the path specified in the command line

- If the path is not specified either in the **#include** statement or on the command line, QRS searches the file relative to the value of the *stp_file_path* ToolInfo variable

Using **#include** in a script is analogous to the way that the C language includes header files (*.h* files) in a body file (*.c* file). However, unlike C header files, no special processing is associated with included files in QRL. Although there is no enforced naming convention for script names, it is recommended that you use a *.qrl* extension for scripts that contain the main function, and a *.inc* extension for included files.

The file being included can also include other files. If the files have duplicate **#include** commands, the duplicate file inclusions are ignored.

Using the #include_if_exists Command

The **#include_if_exists** command is identical to the **#include** command except that it does not return an error message if the target file is not found.

The syntax for including a file is:

```
#include_if_exists "<filename>"
```

For example:

```
#include_if_exists "/qrl/include/optional.inc"
```

For more information, refer to the previous section, "Using the #include Command."

Constructed Data Types and Related Functions

In addition to primitive data types and PDM types, QRL supports the use of structures, enumeration types, and the QRL abstract types list, set, and graph.

User-Defined Types

There are two different kinds of user-defined types: structures and enumeration types.

Structures

Structures are user-defined composite types, which are similar to structs in C and records in other languages. Composite types are useful when you want to group different types of objects together as a single data record. For example, a record in a database might consist of a department number, name, and address.

Structure type definitions must be globally scoped.

Unlike a block of statements, the definition of a structure is bounded by curly braces and ends with a semi-colon.

The syntax for a QRL structure is:

```
struct <name>
{
    <type> <field>;
    <type> <field>;
    ....
};
```

The following example defines a structure called *key* that includes a node *object*, a string *my_name*, and an integer *status*:

```
struct key
{
   node object;
   string my_name;
   int status;
};
```

Variables of a struct type are declared in the same way as other variables:

```
<struct_type> <variable>;
```

For example:

```
key key_var;
```

Variables of struct types are passed by reference when used as parameters to functions:

A field is declared as follows:

```
<variable>.<field>
```

For example:

```
key_var.status;
```

Variables of struct types can be operated on like any other variable.

The script in Figure 12 shows the definition of a structure date that contains the fields “month,” “day,” and “year.” A variable “today” is declared to be of the type date, and values are assigned to the fields contained within “today.” Finally, the contents of “today” are printed.

Figure 12: Sample Script: Using Structures

```
//
// Globally define a struct called date
//
struct date
{
   string month;
   int day;
   int year;
};
void
main()
{
//
// Declare the variable today to be of the type date
//
   date today;
//
// Assign values to the fields today.month, today.day,
// and today.year
//
   today.month = "December";
   today.day = 22;
   today.year = 1998;
//
// Print the values contained in the structure today
//
```

```
    print_line("Today's date is " + today.month + "-" +
                  today.day + "-" + today.year);
}
```

The output of this script is:

```
Today's date is October-22-1998
```

You can also nest structures. For example, the structure date used in the above script could itself be included in another structure:

```
struct time_and_date
{
    time time_struct;
    date date_struct;
};
```

You could then define variables to be of the type time_and_date:

```
time_and_date event;
```

Elements inside a structure contained in another structure are declared as follows:

```
<variable>.<field>.<field>...
```

For example, this statement sets the "month" element of the structure "date_struct" contained within "event" (which is a time_and_date structure variable) to "August":

```
event.date_struct.month = "August";
```

Enumeration Types

QRL supports enumeration types. Using an enumeration type allows you to specify a list of permissible values that can be assigned to the type, such as True and False. This ensures that attempts to assign to a variable a value that is not permitted by the enumeration type fails.

Enumeration type definitions must be globally scoped.

The syntax is:

```
enum <type> {<element_1>,<element_2>.... <element_n>};
```

For example, the following statement defines the values that are valid for an enumeration type Days:

```
enum Days{ Sunday, Monday, Tuesday, Wednesday, Thursday,
   Friday, Saturday };
```

You declare a variable for an enumeration type just as you would declare any variable. For example, you could declare variables to be of the type Days as follows:

```
Days day1, day2, day3;
```

Attempts to assign the variables “day1,” “day2,” and “day3” a value other than Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, or Saturday would fail.

All comparison operations (== != < <= > >=) apply to enumeration types.

The script in Figure 13 defines the enumeration type Days. It then uses four of the QRL built-in enumeration type functions (enum_first, enum_next, enum_last, and enum_prev) to refer to values in Days according to their relative position in the list.

Figure 13: Sample Script: Using Enumeration Types

```
enum Days
{
   Sunday,
   Monday,
   Tuesday,
   Wednesday,
   Thursday,
   Friday,
   Saturday
};
void
main()
{
   Days day1, day2, day3, day4;
   day1 = enum_first("Days");
   day2 = enum_next(day1);
   day3 = enum_last("Days");
   day4 = enum_prev(day3);
   print_line("The first day of the week is " + day1);
   print_line("The day after that is " + day2);
```

```
    print_line("The last day of the week is " + day3);
    print_line("The day before that is " + day4);
}
```

The output of this script is:

```
The first day of the week is Sunday
The day after that is Monday
The last day of the week is Saturday
The day before that is Friday
```

If you tried to refer to a value outside of the range, such as:

```
...
day1 = enum_first("Days");
day2 = enum_prev(day1);
print_line("The day before "+day1+" is "+day2);
...
```

the output would be:

```
The day before Sunday is NULL
```

Table 8 describes the enumeration type functions.

Table 8: Enumeration Type Functions

Function	Return Type	Description
enum_prev (<enum_type> <element>)	<enum_type>	Returns the previous <element> of the enumeration type or NULL if the <element> is the first of this enumeration type.
enum_next (<enum_type> <element>)	<enum_type>	Returns the next <element> of the enumeration type or NULL if the <element> is the last of this enumeration type.

Table 8: Enumeration Type Functions (Continued)

Function	Return Type	Description
enum_last (string <enum_type_name>)	<enum_type>	Returns the last element of the enumeration type.
enum_first (string <enum_type_name>)	<enum_type>	Returns the first element of the enumeration type.
to_enum(string <value>, string <enum_type_name>)	<enum_type>	Returns an element named <value> of the type <enum_type_name>. Generates an error if the value does not correspond to an element of type <enum_type_name>.
to_enum(int <value>, string <enum_type_name>)	<enum_type>	Returns an element of type <enum_type_name> whose position is <value>. Generates an error if <value> is out of range for <enum_type_name>.
to_string (<enum_type> <element>)	string	Returns the name of the <element>.
to_int(<enum_type> <element>)	int	Returns the position of <element> (as in C, elements are numbered starting with 0).

QRL Abstract Types

The QRL abstract types are list, set, and graph. Lists, sets, and graphs are collections of elements.

Lists and sets share the following characteristics:

- They can be homogeneous (that is, have elements that are all the same data type) or heterogeneous.

This cannot apply to graphs, since the elements of graphs (nodes or edges) can represent only strings, whereas elements in a list or set can represent any QRL type.

- Elements are added by copy; an element itself can be a list or set.

  When you copy a graph, all the nodes (strings) and edges are copied to the new graph.

- If an element is a list or set type, a deep copy is added.

  That is, when you add an abstract type to a list or set, all of its elements are copied too. It is a copy of the original list or set; any changes made to the original list or set are not reflected in the copy (and vice-versa).

  This cannot apply to graphs, since they cannot contain other types.

- Lists and sets are passed by reference when used as parameters to functions.

  This is also true of graphs.

Unlike using arrays in C, memory management happens automatically when you use lists, sets, and graphs in QRL. You do not need to allocate and deallocate space when you create a list, set, or graph, or when you add or delete elements.

Lists and sets differ in the following ways:

- Sets cannot include duplicate values, lists can.
- The order of elements is guaranteed in lists, but not in sets.

  Lists are indexed 0 to last, where last is (list_size - 1). New elements are appended to the end of the list, which is not necessarily true for sets.

- Sets have the standard set operations—intersection, union, and difference.

Lists are used more commonly than sets for performance reasons; sets are generally used only when there is a need to eliminate duplicate values or perform set operations such as union and intersection.

The list Type

A list is an ordered, indexed collection of objects that is comparable to an array or a linked list in C.

The syntax for declaring a list is:

```
list <list_name>;
```

For example:

```
list my_list;
```

Lists can either be typed (homogeneous) or untyped (heterogeneous). The following examples show the creation of an empty typed list (typed_list) that can contain only integers, and an empty untyped list (untyped_list) that can contain elements of any data type:

```
typed_list = list_create("int",0);
untyped_list = list_create("all",0);
```

The 0 parameter in the list_create statement indicates that the initial size of the list is 0.

The script in Figure 14 sorts a list of integers into ascending order. The script contains three functions: main, sort, and print_list. The functions sort and print_list are called by main, and both take two arguments: a list and the number of elements in the list.

The sort function uses a set of nested for loops. The outermost loop sequences through the list from the first element to the next-to-last element. For each element, a second for loop is entered, which starts from the element after the one currently selected by the outer loop, and ranges through the last element of the list.

If the elements are out of order (that is, if first < second), then the elements are switched.

Figure 14: Sample Script: Using Lists

```
void main()
{
    list my_list;
    int count;
//
// Create a list
//
    my_list = list_create("int", 0);
//
// Add elements to the list
```

```
//
   list_append(my_list, 3);
   list_append(my_list, 5);
   list_append(my_list, 10);
   list_append(my_list, 2);
//
// Store length of list in variable count
//
   count = list_count(my_list);
//
// Print list before sort using print_list function
//
   print_line("The list before the sort is:");
   print_list(my_list, count);
   print_line("\n");
//
// Call sort function with list and list count
//
   sort (my_list, count);
//
// Print list after sort using print_list function
//
   print_line("The list after the sort is:");
   print_list(my_list, count);
}

//
// sort function
//
void sort (list sort_list, int n)
{
   int i, j, first, second;
//
// Use nested for loops to compare the values i and j
// and swap them if i > j
//
   for ( i = 0; i < n; i = i + 1){
      for ( j = i + 1; j < n; j = j + 1 ){

         first = list_get(sort_list, i);
         second = list_get(sort_list, j);

         if (first > second)
         {
            list_set(sort_list, i, second);
```

```
            list_set(sort_list, j, first);
         }
      }
   }
}
//
// print_list function
//
void print_list ( list pr_list, int length )
{
   int i, current_value;
   for (i = 0; i < length; i = i + 1){
      current_value = list_get(pr_list, i);
      print(current_value +" ");
   }
}
```

The output from this script is:

```
The list before the sort is:
3 5 10 2
The list after the sort is:
2 3 5 10
```

Both the sort and print_list functions have the return type void. This is particularly significant in the case of the sort function, which changes the list and yet does not return the modified list to main. This is because list, set, and graph objects are passed by reference. That means that if you pass a list or set into a function and make changes to the formal parameter within the function (by using list_clear, list_set, and so on), the actual parameter (the original list or set) is modified as well. Consequently, sort does not need to return the list after it has modified it, because it modifies the original list directly.

However, assigning a list, set, graph, or single element to a variable makes a copy of the original. Subsequently changing the value of the variable does not cause a corresponding change in the original list, set, graph, or element. So, for example, to change the value of an element in a list that has been assigned to a variable, the list needs to be explicitly updated using the list_set function. For an example, see "Passing list, set, and graph Values" on page 3-89.

For more discussion of the difference between pass by value and pass by reference, see "Calling Functions" on page 3-29 and the subsequent script example, Figure 11 on page 3-30.

Table 9 describes the built-in functions provided for the list type.

When a type is listed as "all," it means that it can be any data type supported by QRL, including user-defined types.

Table 9: QRL list Functions

Function	Return Type	Description
list_append(list <the_list>, <any type> <item>)	int	Appends an item to the end of <the_list>, and returns <the_list> length. If <item> is of the wrong type, an error is flagged.
list_concatenate(list <list1>, list< list2>)	int	Concatenates a deep copy of <list2> to <list1> and stores the result in <list1>. Returns <list1> length. If the lists are not of the same type, an error is flagged.
list_create(string <member_type>, int <initial_size>)	list	Creates a list whose items are of <member_type>. If "all" is used as the value of <member_type>, the list is heterogeneous; if <initial_size> > 0, the list is created to be of that size, and padded with NULLs.

Table 9: QRL list Functions (Continued)

Function	Return Type	Description
list_delete(list <the_list>, int <index>)	int	Deletes the item at position <index> in <the_list>. Items <index> + 1 through last become items <index> through last - 1. Returns list length. If <index> > last or < 0, an error is flagged; unused space is freed.
list_get(list <the_list>, int <index>)	all	Returns the value of the item at position <index> in <the_list>. If <index> is out of range, an error is flagged.
list_get_type(list <the_list>, int <index>)	string	Returns the type of the item at position <index> in <the_list>. If <index> is out of range, an error is flagged.
list_set(list <the_list>, int <index>, all value)	void	Sets the item at position <index> in <the_list> and frees the previous item at <index>. If <index> is out of range or the value is of the wrong type, an error is flagged.
list_to_string(list <the_list>, string <separator>)	string	Converts a list to a string with the <separator> string separating the string representation of each item in <the_list>.
string_to_list(string the_string, string <separator>s)	list	Converts a string to a list; <the_string> is broken into tokens separated by one of the characters in <separator>.
lists_equal(<list1>, <list2>)	boolean	Tests whether two lists have the same type and the same values in the same order.

Table 9: QRL list Functions (Continued)

Function	Return Type	Description
list_count(list <the_list>)	int	Returns list length.
list_insert(list <the_list>, int <index>, <all item>)	int	Inserts an item before <index>. Items <index> through last become items (<index> + 1) through (last + 1). Returns list length. If <index> > last or < 0, an error is flagged. If <item> is of the wrong type, an error is flagged.
list_find(list <the_list>, int <start_index>,< all item>)	int	Returns index of <item> in <the_list>. The search starts at <start_index>; returns count if not found. An error is flagged if the item type is not the same as the type stored in <the_list>, or if <start_index> is out of range.
list_copy(list <the_list>)	list	Returns a copy of <the_list>.
list_clear(list <the_list>)	void	Clears <the_list> and frees the space.
list_select (string <oms_query>)	list	Executes <oms_query> and returns a list containing the results. The list type corresponds to the query.

Enumeration Types and list Functions

All of the list function calls that take an index as a parameter can be called with an enumeration type value.

For example, the function list_find returns an index. If list_find is called with an enumeration type object as the <start_index> parameter, an enumeration type object is returned. If the search object is found at an

index position that is out of range for the enumeration type, an error is generated. If the search item is not found in the list, NULL is returned.

The script in Figure 15 demonstrates how you can use enumeration types as an index into a list. It defines a structure `thing` that includes an object name and type, and an enum `Types` that defines those types. It uses a subfunction set_master_list to assign objects a type and read them into a list of lists that is then used by the main function. The main function prints the list.

Figure 15: Sample Script: Using an Enumeration Type As Index

```
enum Types
{
   Animal,
   Vegetable,
   Mineral
};
struct thing
{
   string name;
   Types type;
};
string input = "dog cat hamster rabbit:" +
       "carrot rutabaga tomato lettuce:" +
       "granite bauxite quartz iron";
string element_sep = " ";
string list_sep = ":";
// Sets up and prints a master list of the input.
//
void main ()
{
   list master_list;
   master_list = set_master_list(input, list_sep,
      element_sep);
   print_line("Master list:");
   print_line(master_list);
}
// Converts a string of strings of elements into a list
// of things.
// Assumes the strings of elements are in the order of
// the enum Types.
//
list set_master_list(string input, string list_sep,
```

```
        string element_sep)
{
    int ix, count, ix2, count2;
    list master_list, thing_list, current_type_list,
        list_by_type;
    Types current_type;
    thing current_thing;
    master_list = list_create("list", 0);
    thing_list = list_create("thing", 0);
    list_by_type = string_to_list(input, list_sep);
    // Simultaneously cycle through the list of strings
    // and the Types enum.
    // If we exhaust strings before Types, pad the master
    // list with NULLs.
    // If we exhaust types before strings, inform the user.
    //
    for (ix = 0, current_type = enum_first("Types"),
        count = list_count(list_by_type);
        current_type != NULL;
        current_type = enum_next(current_type))
    {
    // If we've exhausted the list of strings, just want
    // to add a NULL for each extra type.
    //
        if (ix == count)
        {
            list_append(master_list, NULL);
            continue;
        }
        current_type_list =
        string_to_list(list_get(list_by_type, ix),
            element_sep);
        for (ix2 = 0, count2 = list_count(current_type_list);
            ix2 < count2; ix2 = ix2 + 1)
        {
            current_thing.name = list_get(current_type_list,
                ix2);
            current_thing.type = current_type;
            list_append(thing_list, current_thing);
        }
        list_append(master_list, thing_list);
        list_clear(thing_list);
        // Note that we want to increment this index only if
        // we didn't continue above.
        ix = ix + 1;
```

```
    }
    if (ix != count)
    {
        print ("This input was not omitted from the master " +
            "list of lists:\n");
        for (ix2 = ix; ix2 < count; ix2 = ix2 + 1)
        print(list_get(list_by_type, ix2));
        print_line();
    }
    return master_list;
}
```

The output from this script is:

```
Master list:

{
 0 =
    {
     0 =
        {
         name = dog
         type = Animal
        }
     1 =
        {
         name = cat
         type = Animal
        }
    . . .
    }
 1 =
    {
     0 =
        {
         name = carrot
         type = Vegetable
        }
     1 =
        {
         name = rutabaga
         type = Vegetable
        }
    . . .
```

```
   }
 2 =
   {
    0 =
      {
       name = granite
       type = Mineral
      }
    1 =
      {
       name = bauxite
       type = Mineral
      }
   . . .
```

Note that unlike Figure 14 on page 3-41, which populated a list by appending each individual element, this script uses the string_to_list function to create and populate a list.

The set Type

A set is a collection of unique objects. You can perform standard set operations on a set such as union, intersection, and difference.

The syntax for declaring a set is:

```
set <set_name>;
```

For example:

```
set my_set;
```

Like lists, sets can either be typed (homogeneous) or untyped (heterogeneous). The following statement creates a set, “my_set,” that can contain all data types:

```
my_set = set_create("all");
```

The script in Figure 16 defines a function list_func that uses the function string_to_list to create a list of strings. The function list_func is called by the main function, which provides arguments for list_func, creates a set my_set of the type string, and attempts to copy the elements of my_list into my_set.

Because a set can only contain unique values, duplicate values are rejected. This script flags duplicate elements in my_list, adds unique values from my_list to my_set, and prints the contents of my_list and my_set.

Figure 16: Sample Script: Using Sets

```
void
main()
{
   set my_set;
   list my_list;
   string current_value;
   int count, ix;
   // Create a set to contain strings
   my_set = set_create("string");
   // populate my_list using function list_func
   my_list = list_func("apples oranges potatoes apples");
   count = list_count(my_list);
   for (ix = 0; ix < count; ix = ix + 1)
   {
      // Fetch element from my_list; test to see if it is
      // already a member of the set
      current_value = list_get(my_list, ix);
      if (set_is_member(my_set, current_value))
            print_line("Duplicate value \""+current_value+
                  "\" is element "+(ix+1)+" of list \n");
      // If element fetched from my_list is not a member
      // of my_set,add it to the set
      else
            set_add(my_set, current_value);
   }
   // Print the contents of my_set
   print_line("The elements in the set are:");
   print_line(my_set);
}
// Function list_func; returns a list
list
list_func(string input)
{
   list my_list;
   // Use string_to_list function to create a string list
   // from the parameter input
   my_list = string_to_list(input, " ");
```

```
   // Print the contents of my_list
   print_line("The elements in the list are:");
   print_line(my_list);
   print_line("\n");
return my_list;
}
```

The output from the script is:

```
The elements in the list are:
{
   0 = apples
   1 = oranges
   2 = potatoes
   3 = apples
}
Duplicate value "apples" is element 4 of list
The elements in the set are:
{
   0 = apples
   1 = oranges
   2 = potatoes
}
```

Note that this script uses the print_line function to print the contents of my_list and my_set without explicitly fetching each element. Using print_line to display the contents of a list or set is useful when you do not need to format the output.

Table 10 describes the set functions.

Table 10: QRL set Functions

Function	Return Type	Description
set_create (string <member_type>)	set	Creates an empty set whose elements are of type <member_type>. If "all" is used as the value of <member_type>, the set is heterogeneous.

Table 10: QRL set Functions (Continued)

Function	Return Type	Description
set_add(set <the_set>, <all element>)	int	If <element> is not a member of <the_set>, adds <element> to the set and returns <the_set> cardinality (size). If <element> is of the wrong type, an error is flagged.
set_is_member(set <the_set>, <all element>)	boolean	Returns True if <element> is in <the_set>, False otherwise.
set_delete(set <the_set>, <all element>)	int	If <element> is a member of <the_set>, delete <element>. Returns <the_set> cardinality (size).
set_clear(set <the_set>)	void	Empties <the_set>.
set_copy(set <the_set>)	set	Returns a deep copy of <the_set>.
set_count(set <the_set>)	int	Returns the cardinality of <the_set>.
sets_equal(set <set1>, set <set2>)	boolean	Tests whether two sets have the same type and elements. Returns True or False.
set_union(set <set1>, set <set2>)	int	Stores the union of <set1> and <set2> into <set1>; returns the cardinality of <set1>. If sets are not of the same type, an error is flagged.
set_intersection(set <set1>, set <set2>)	int	Stores the intersection of <set1> and <set2> into <set1>; returns the cardinality of <set1>. If sets are not of the same type, an error is flagged.

Table 10: QRL set Functions (Continued)

Function	Return Type	Description
set_difference(set <set1>, set <set2>)	int	Stores the difference of <set1> and <set2> into <set1>; returns the cardinality of <set1>. If sets are not of the same type, an error is flagged.
set_is_subset(set <set1>, set <set2>)	boolean	Returns True if <set2> is a subset of <set1>.
set_is_proper_subset (set <set1>, set <set2>)	boolean	Returns True if <set2> is a proper subset of <set1>.
set_select(string <oms_query>)	set	Executes <oms_query> and returns a set containing the result. The returned set is of the type corresponding to the query.
set_get_element(set <the_set>, int <nth>)	all	Returns the value of the <nth> element in <the_set>. If <nth> < 0 or >= to <the_set> cardinality, an error is flagged.
set_get_type(set <the_set>, int <nth>)	string	Returns the type of the <nth> element in <the_set>. If <nth> < 0 or >= <the_set> cardinality, an error is flagged.

The graph Type

A “graph” is an object made up of nodes and edges.

A “node” is identified by a string label. Node labels are unique; it is guaranteed that only one node of a particular label exists in the graph.

An “edge” is a directed relationship between nodes. Edges are unique; it is guaranteed that only one edge (directed) between a particular pair of nodes exists in the graph.

The syntax for declaring a graph is:

```
graph <graph_name>;
```

For example:

```
graph my_graph;
```

Unlike lists and sets, graphs can not be heterogeneous. The following statement creates a graph, “my_graph”:

```
my_graph = graph_create();
```

You can use a graph’s nodes and edges to represent nodes and links in a repository. This leads to many useful applications of this data structure with the OMS / PDM data model.

For example, assume you want to check whether there are cycles in a class’s generalization hierarchy in StP / UML. One method is to issue queries to the repository for:

- The class
- The class’s superclass relationship links
- Each superclass link’s superclass
- Each superclass’s superclass relationship links
- The superclass for each of these links

And so on and so on. All these individual queries take time.

To perform the same task using graphs, you load all the class names from the repository into a graph, each class being represented as a node in the graph, then establish an edge in the graph for each pair of nodes that participate in a class / superclass relationship. By using the graph_is_acyclic() function, you can immediately find out if there are any cycles in any of the generalization relationships.

Table 11 describes the graph functions.

Table 11: QRL Graph Functions

Function	Return Type	Description
graph_create()	graph	Creates an empty graph.

Table 11: QRL Graph Functions (Continued)

Function	Return Type	Description
graph_create(set <the_set>)	graph	Creates a graph from the set of strings (that is, it creates a graph of *N* nodes, whereby each node is given a label from the set of strings).
graph_create(list <the_list>)	graph	Creates a graph from the list of strings (that is, it creates a graph of *N* nodes whereby each node is given a label from the list of strings).
graph_clear (graph <the_graph>)	none	Clears <the_graph>.
graph_copy (graph <the_graph>)	graph	Returns a copy of <the_graph>.
graph_count (graph <the_graph>)	int	Returns the number of nodes in <the_graph>.
graph_add_node (graph <the_graph>, string <label>)	none	Adds a node with label to <the_graph> (if it does not already exist).
graph_is_node (graph <the_graph>, string <label>)	boolean	True if a node with a label exists in <the_graph>, otherwise False.
graph_all_nodes (graph <the_graph>)	list	Returns a list of strings of the labels of all the nodes in <the_graph>.
graph_add_arc (graph <the_graph>, string <from_label>, string <to_label>)	none	Adds an edge from node <from_label> to node <to_label>. If either node does not exist, then they are implicitly added.

Table 11: QRL Graph Functions (Continued)

Function	Return Type	Description
graph_is_arc (graph <the_graph>, string <from_label>, string <to_label>)	boolean	True if an edge from node <from_label> to node <to_label> exists, otherwise False.
graph_is_indirect_arc (graph <the_graph>, string <from_label>, string <to_label>)	boolean	True if there is a direct / indirect edge from node <from_label> to node <to_label> exists, otherwise False.
graph_node_predecessors (graph <the_graph>, string <to_label>)	list	Returns a list of strings of the labels of all the nodes that have an edge going to node <to_label>.
graph_node_successors (graph <the_graph>, string <from_label>)	list	Returns a list of strings of the labels of all the nodes that have an edge coming from node <from_label>.
graph_node_indirect_ predecessors (graph <the_graph>, string <to_label>)	list	Returns a list of strings of the labels of all the nodes that have a direct / indirect edge going to node <to_label>.
graph_node_indirect_ successors(graph <the_graph>, string <from_label>)	list	Returns a list of strings of the labels of all the nodes that have a direct / indirect edge coming from node <from_label>.
graph_node_rank (graph <the_graph>, string <label>)	int	Returns the rank of node labelled <label>.
graph_topological_sort (graph <the_graph>)	list	Returns a list of strings of the labels of all the nodes in <the_graph> in topological order (that is, by rank).
graph_is_acyclic (graph <the_graph>)	boolean	True if the graph has no cycles, otherwise False.

String Manipulation Functions

QRL includes built-in string manipulation functions that enable you to:

- Extract strings
- Find and replace strings
- Change the case of strings
- Return the length of strings

The script in Figure 17 illustrates the use of the built-in string manipulation functions string_length, string_find, string_extract, to_upper, and to_lower.

Figure 17: Sample Script: String Manipulation Example

```
void
main()
{
   const int LENGTH = 9;
   string s1 = "This is a character string.";
   int i;
   string s2;
   // Print the string s1
   print_line("Test string: " + s1 + "\n");
   // Return the length of the string s1
   print_line("String length: " + string_length(s1) + "\n");
   // Search for a specified character string;
   // return location
   i = string_find(s1, 0, ".");
   print_line("The string \".\" is at position " + i +
      " in the string. \n");
   // Extract character string of length LENGTH starting
   // at position 0
   s2 = string_extract(s1, 0, LENGTH);
   print_line("First " + LENGTH + " characters of " +
      "the string: \"" + s2 + "\"\n");
   // Change character string to uppercase
   s2 = to_upper(s1);
   print_line("Change to uppercase: " + s2 + "\n");
   //Change character string to lowercase
   s2 = to_lower(s1);
```

```
    print_line("Change to lowercase: " + s2);
}
```

The output from this script is:

```
Test string: This is a character string.
String length: 27
The string "." is at position 26 in the string.
First 9 characters of the string: "This is a"
Change to uppercase: THIS IS A CHARACTER STRING.
Change to lowercase: this is a character string.
```

The last character in the string (.) is at position 26, even though the string has a length of 27. This is because the string is numbered 0-n.

Table 12 lists the string manipulation functions available in QRL.

Table 12: String Manipulation Functions

Function	Return Type	Description
string_search_and_replace (string <the_string>, string <search_for>, string <replace_by>)	string	Returns a string with all occurrences of <search_for> in <the_string> replaced by the <replace_by> string.
string_extract (string <the_string>, int <start>, int <length>)	string	Extracts the substring starting at <start> of <length>.
string_find (string <the_string>, int <start>, string <the_substring>)	int	Returns the index into <the_string> where <the_substring> occurs. Starts the search at <start>. Returns the length of <the_string> if <the_substring> is not found.
string_length (string <the_string>)	int	Returns the length of <the_string>.
to_lower(string <the_string>)	string	Makes <the_string> lowercase.
to_upper(string <the_string>)	string	Makes <the_string> uppercase.

Table 12: String Manipulation Functions (Continued)

Function	Return Type	Description
string_converts (string <the_string>, string <target_type>)	boolean	Tests whether the string legally converts to a value of <target_type> (<target_type> must be one of the primitive types).
string_escape (string <the_string>, string <badchars>)	string	Escapes specified <badchars> in <the_string> and converts newlines to \ \n. Returns string with <badchars> escaped.
string_translate (string <the_string>, string <input_table>, string <output_table>)	string	Translates <the_string> by substituting the value specified in <output_table> for corresponding value specified in <input_table>. Returns translated string.
string_strip (string <the_string>, string <option>, string <stripchars>)	string	Strips specified <stripchars> according to the value specified for <option>: L (leading), T (trailing), or B (both). Returns string with <stripchars> removed.

File Functions

QRL includes built-in file functions that enable you to perform Input/Output (I/O) operations. These functions allow you to:

- Read, write, delete, and copy files
- Return path and file names
- Return read and write permissions on files

The read_file, write_file, and delete_file Functions

The read_file function allows a file to be read. The syntax of the read_file function is:

```
string read_file(string <file_name>)
```

This returns the contents of <file_name>.

The write_file function writes to a file and returns the number of characters written to the file. The syntax is:

```
int write_file(string <path>, string <contents>)
```

The delete_file function deletes a specified file. This is useful for any **qrp** scripts that create temporary files and need to delete them. The syntax of the delete_file function is:

```
void delete_file(string <file_name>)
```

List of File Functions

The file functions are described in Table 13.

Table 13: File Functions

Function	Return Type	Description
append_file(string <the_file>, string <string_to_add>)	void	Appends <string_to_add> to <the_file>.
copy_file(string <from>, string <to>)	int	Copies file <from> to file <to>. Returns -1 upon failure.
delete_file(string <file_name>)	void	Deletes the <file_name> file.
directory_files (string <directory>, string <pattern>)	list	Returns a list of files in <directory> matching <pattern>. If <pattern> is NULL or "", "*" is used as the pattern.

Table 13: File Functions (Continued)

Function	Return Type	Description
file_exists(string <the_file>)	boolean	True if <the_file> exists, otherwise False.
file_part(string <path>)	string	Returns the filename part of <path>. If <path> has no file_part, "" is returned.
home_directory(void)	string	Returns the user's home directory path.
mkdir(string <directory>)	int	Makes a directory named <directory>. Returns -1 upon failure.
parent_directory (string <path>)	string	Returns the parent directory of <path>.
path_compose (list <components>)	string	Returns a path composed of components. Some normalization is done so that there is exactly one delimiter between components.
path_compose (string <component1>, string <component2>....)	string	Returns a path composed of components. Some normalization is done so that there is exactly one delimiter between components.
path_last_delimiter (string <path>)	int	Returns the position in <path> of the last path delimiter character. The length of the string <path> is returned if there are no delimiter characters.
path_normalize(string <path>)	string	Makes all delimiters in <path> look the same.

Table 13: File Functions (Continued)

Function	Return Type	Description
path_part(string <path>)	string	Returns the path prefix of <path> including the final path delimiter. If <path> has no path part, "" is returned.
read_dir_access (string <the_dir>)	boolean	True if the directory can be read, otherwise False.
read_file(string <file_name>)	string	Returns the contents of <file_name>.
read_file_access (string <the_file>)	boolean	True if the file can be read, otherwise False.
root_part(string <path>)	string	Returns the root part of <path>. If <path> has no root part, NULL is returned.
write_file(string <path>, string <contents>)	NULL	Writes <contents> to a file (<path>) and returns the number of characters written to the file.
write_file_access (string <the_file>)	boolean	True if the file can be written, otherwise False.

System Functions

QRL includes built-in system functions that enable you to:

- Send output messages to a shell window while running a script
- Return information from your system, such as a user name or the time
- Issue operating system commands using the “system” function

The message Function

One of the system functions available in QRL is the message function. The message function sends output to stderr (standard error) while a script is running. The print functions, on the other hand, send output after the script finishes running; the output does not necessarily go to stderr, and can include formatting information.

The syntax of the message function is:

```
void message(<all types> <value>);
```

For example:

```
...
for (x = 0; x < n; x = x + 1)
{
   <...>
   message("The value of x is " + x);
}
...
```

Using the message function in this for loop outputs the value of x to stderr every time the loop is executed.

The message function is particularly useful in testing a script or providing feedback to users during document generation. The document generation process can be time-consuming, particularly for large scripts and large systems, so it is helpful to inform the user with messages as the generation proceeds.

When a script is run from the command line, the output from the message function is sent to the message window (assuming that it is defined as stderr). If you create a document from the Script Manager, you must choose **Show Message Log** from the **View** menu in order to see the messages displayed in the Message Log window.

Functions for Returning System Information

QRL includes built-in functions to return information about the system.

This sample script prints out the name of the user and the current time.

```
void
main()
{
    print_line("The user is " + user());
    print_line("The current time is " +
        time_to_string(time_now(), NULL));
}
```

The output from this script resembles the following:

```
The user is smith
The current time is 04/27/98 14:08:37
```

Function for Issuing Operating System Commands

The “system” function provides a mechanism for issuing operating system commands from a script. The syntax is:

```
void system(string <operating_system_cmd>)
```

The command is passed without being interpreted to the operating system, with one exception—the command is scanned for variable substitution syntax (both script and ToolInfo variables), and if it is detected, the variables are replaced with their respective values in the command.

For more discussion of variable substitution syntax, see Chapter 4, “The OMS Query Language and QRL.”

List of System Functions

The system functions are described in Table 14.

Table 14: System Functions

Function	Return Type	Description
message(<all value> <value>)	void	Outputs value to stderr.

Table 14: System Functions (Continued)

Function	Return Type	Description
message(<all value> <value>, boolean <CR>)	void	Outputs value to stderr. Outputs to same line when <CR> is False, otherwise True.
toolinfo_variable (string <toolinfo_var>)	string	Returns the value of <toolinfo_var>.
user()	string	Returns username.
hostname()	string	Returns hostname.
current_system()	string	Returns the current system (may be different from the value of the *system* ToolInfo variable if that variable has been overridden).
current_projdir()	string	Returns the current project directory (may be different from the value of the *projdir* ToolInfo variable if that variable has been overridden).
time_now()	int	Returns the number of seconds since 1970 in Greenwich Mean Time.
time_to_string(int <time>, string <fmt>)	string	Returns string representation of <time>; <fmt> parameter works the same way C-library <strptime fmt> parameter works. If <fmt> is NULL, a default is used.
string_to_time(string <time>, string <fmt>)	int	Returns integer representation of <time>; <fmt> parameter works the same way C-library <strftime fmt> parameter works. If <fmt> is NULL, a default is used.

Table 14: System Functions (Continued)

Function	Return Type	Description
system(string <operating_system_cmd>)	void	Executes operating system command.

Administrative Functions

The adminstrative functions described here fall into two categories:

- Repository functions, which manage repository administrative tasks
- System repository functions, which operate on an StP system and its repository

With both of these types of functions, there are four discretionary security groups: system administrator, repository (database) owner, system writers, and system readers. The default group is system writers. If the system administrator has a password, security is enforced. If not, all system users can execute any command.

For password information, see *StP Administration*.

With these functions, you can specify default values for the parameters that are enclosed in curly { } braces, using the following conventions:

Table 15: Conventions for Setting Defaults

Parameter Type	Use This Setting	Example
string	An empty string	`repos_maint("")` Use 0-length strings.
integer or float	Enter “-1”	`sys_expand_rep("c:/work/", "toysys", -1)` The -1 refers to the default int size.
optional_bool_enum	Enter “Default”	`repos_show_space("myserver", Default)`

Repository Functions

The following table describes the repository-specific functions; Table 17 on page 3-70 describes the functions' parameters in detail.

Table 16: Repository Functions

Function	Return Type	Description
repos_add_maker(string <users>, {string <server>}, {string <password>})	int	Grants system creation privileges to one or more Sybase users. Automatically adds the specified users to the server. You do not need to call repos_add_user separately for each new user. For example, you can add a string of multiple users, each user separated by a blank space. Use the <password> parameter if the server is secure or partially secure and the user is a new user of the server. If you do not specify a password, users will be prompted to enter one. See also repos_supports_makers, described in this table.
repos_add_user(string <users>, {string <server>}, {string <password>})	int	Adds one or more Sybase users to the server, which allows them to connect to an StP system. If you do not specify a password, users will be prompted to enter one. See also repos_can_modify_users, described in this table, and sys_can_modify_users, described in Table 18 on page 3-71.
repos_can_list_current_users ({string <server>})	boolean	Returns True if the server can list current users. This function returns True for Sybase and False for Microsoft Jet.
repos_can_modify_users({string <server>})	boolean	Returns True if the server can modify user information. If a server returns False for this function, it will not support repos_add_user, repos_delete_user, and repos_change_password. This function is True for Sybase and False for Microsoft Jet.

Table 16: Repository Functions (Continued)

Function	Return Type	Description
repos_change_password(string <users>, {string <server>}, {string <password>})	int	Changes one or more Sybase server users' passwords to the specified password. Users who do not have administrative privileges can only use repos_change_password to change their own passwords. See also repos_can_modify_users, described in this table, and sys_can_modify_users, described in Table 18 on page 3-71.
repos_delete_maker(string <users>, {string <server>})	int	Revokes system creation privileges from one or more Sybase users. See also repos_supports_makers, described in this table.
repos_delete_user(string <users>, {string <server>})	int	Removes users from the server. Users can no longer connect to StP systems on a server from which they have been removed. repos_delete_user also removes the users from any systems they formerly had access to. You do not need to call sys_delete_user separately. See also repos_can_modify_users, described in this table, and sys_can_modify_users, described in Table 18 on page 3-71.
repos_has_dynamic_space({string <server>})	boolean	Returns True if databases on the server will expand automatically as necessary; otherwise, it returns False. If this function returns True for a server, that system will not support the repos_show_space function. All systems on that server will not support sys_expand_rep or sys_show_space. This function returns False for Sybase and True for Microsoft Jet.
repos_list_reps({string <server>})	int	Lists all repositories under the server.
repos_list_users({string <server>})	int	Displays the names of all valid server users. These users do not need to be currently using the server.

Table 16: Repository Functions (Continued)

Function	Return Type	Description
repos_list_current_users({string <server>})	int	Displays the names of all valid users currently using the server.
repos_maint({string <server>})	int	Performs routine server maintenance, including deleting old temporary tables (Microsoft Jet and Sybase), dumping the server's transaction log (Sybase only), and updating the statistics used by the Adaptive Server or SQL Server indexes and optimizer.
repos_show_space({string <server>}, {optional_bool_enum <all>})	int	Shows the available space, in megabytes, in the Sybase repository. The <all> parameter shows space for all of the devices in the Sybase server. See also the repos_has_dynamic_space function, described in this table.
repos_supports_makers({string <server>})	boolean	Returns True if the server supports repos_add_maker and repos_delete_maker. This function is True for Sybase and False for Microsoft Jet.

Parameters

Table 17: Repository Function Parameters

Parameter	Description
<server>	The name of the server; the default is the value of the DSQUERY environment variable. If DSQUERY is not set, StP assumes the repository is Microsoft Jet.

Table 17: Repository Function Parameters

Parameter	Description
<password>	The password to be assigned to a user or users; the default is blank (no password). By default, StP first checks the <server>_PASSWORD environment variable for the password, if the function is referring to the Sybase user password. If it is not set there, it then checks the ToolInfo file for the *<server>_password* setting, as well as *<server>_admin_password*, *msjet_password*, and *msjet_admin_password* for the other password types. If you do not specify a password, StP prompts the user for one.
<users>	The name of one or more users, which must be valid user names at the operating system level. Multiple user names must be separated by spaces, for example, “larry moe curly”, not “larry, moe, curly”.

System Repository Functions

The system repository functions operate on an StP system and its repository. A system is a set of flat files and repository storage for objects referenced by the flat files. The flat files are grouped by editor types and each file contains symbol information for a diagram or table. Also, each system has a *.repinfo* file that points to the system’s repository storage.

The following table describes the system-specific functions; Table 19 on page 3-78 describes the functions’ parameters in detail.

Table 18: System Functions

Function	Return Type	Description
sys_add_user({string <projdir>}, {string <system>}, {string <writers>}, {string <readers>})	int	Adds Sybase users to a system. In a secure system, the executor must have at least owner permissions to execute this function. See also sys_can_modify_users, described in this table.

Table 18: System Functions (Continued)

Function	Return Type	Description
sys_can_list_current_users({string <projdir>}, {string <system>})	boolean	Returns True if the system can list current users, otherwise False. If False, the server that the system is on will not support repos_list_current_users. This function returns True for Sybase systems and False for Microsoft Jet systems.
sys_can_modify_users({string <projdir>}, {string <system>})	boolean	Returns True if the system can modify user information. If a system returns False for this function, it will not support sys_add_user, sys_delete_user, sys_change_user, and sys_change_repowner. The server that the system is on will not support repos_add_user, repos_delete_user, and repos_change_password, described in Table 16 on page 3-68. This function returns True for Sybase systems and False for Microsoft Jet systems.
sys_change_repowner({string <projdir>}, {string <system}>, string <user>, {string <password>})	int	Changes the owner of the system's Sybase repository and adds the specified new user to the server. You do not need to call repos_add_user separately. Use the <password> parameter to set the new user's password in the server. See also sys_can_modify_users, described in this table. This function is not valid for Microsoft Jet.
sys_change_user ({string <projdir>}, {string <system>}, {string <writers>}, {string <readers>})	int	Changes the Sybase security level between reader and writer users. Using the <writers> parameter grants read and write privileges. Using the <readers> parameter grants read-only privileges. The executor must be at least have owner permissions in a secure system. See also sys_can_modify_users, described in this table.

Table 18: System Functions (Continued)

Function	Return Type	Description
sys_clean_rep({string <projdir>}, {string <system>}, {optional_bool_enum <force>})	int	Deletes all data from the system's repository, but does not destroy the repository. It does not affect files in the system's directory. In a secure system, the executor must be at least an owner. Users must not be running any editors (GDE, GTE, OAE) while this command is running.
sys_create({string <projdir>}, {string <system>}, {string <repositoryType>}) and sys_create({string <projdir>}, {string <system>}, {string <repositoryType>}, {int <size>}, {string <server>}, {string <filemode>}, {string<repname>)	int	Creates a new system, including its *.repinfo* file and directories for its various editors' flat files. The executor, who must be a system administrator (or maker; see repos_add_maker in Table 16 on page 3-68) in a secure environment, becomes the owner of the new system and the database owner of the system repository's underlying database. When a new system is created, locking is automatically enabled, and the system owner is the lock administrator. To disable locking or to change the lock administrator, use the **Locks** submenu available from the **Tools** menu on the Desktop. For Sybase, the default setting for the <size> parameter is 6. By default, sys_create provides an additional 2 MB for the system's transaction log. You can reset the transaction log size with the *syscreate_log_size* ToolInfo variable. The default for <filemode> is c (change).
sys_delete_user({string <projdir>}, {string <system>}, string <users>)	int	Denies Sybase users access to the specified system. In a secure system, the executor must at least have owner permissions. See also sys_can_modify_users, described in this table.
sys_destroy({string <projdir>}, {string <system>}, {optional_bool_enum <force>})	int	Destroys a system's repository and all files in the system directory. The system cannot be regenerated after being destroyed with this command. In a secure system, the executor must at least have owner permissions.

Table 18: System Functions (Continued)

Function	Return Type	Description
sys_destroy_rep({string <projdir>}, {string <system>}, {optional_bool_enum <force>})	int	Destroys a system's repository, but does not remove the files from the system directory. The repository can be regenerated from the files after being destroyed by this command. In a secure system, the executor must at least have owner permissions.
sys_dump_rep({string <projdir>}, {string <system>}, {string <dumpdir>}, {optional_bool_enum <dumpuserinfo>})	int	For Sybase only. Dumps a system's repository to special dump files that can be subsequently reloaded with the sys_load_rep function or from the Desktop **Repository** menu (from the **Repository** menu, choose **Maintain Systems** > **Load Current System Repository from Files**). Overwrites any existing files in the specified dump directory. A repository can be dumped while users are accessing it. The dump reflects the state of the data at the time the dump began. Dump a repository to provide a backup that can be reloaded in the event of media failure, or to copy a repository to another system or another machine. See also sys_load_rep and sys_supports_rep_dump, described in this table.
sys_duplicate({string <oldProjdir>}, string <oldSystem>, {string <newProjdir>}, string <newSystem>)	int	Creates a duplicate copy of an existing system with a new name.

Table 18: System Functions (Continued)

Function	Return Type	Description
sys_expand_rep({string <projdir>}, {string <system>}, {int <size>})	int	For Sybase only. Expands the system repository by allocating additional space for its use on the device specified by the *syscreate_device* ToolInfo variable. The default expansion is 2 MB, and the default device is "default." In a secure system, the executor must be at least the system administrator or have owner permissions. See also the sys_has_dynamic_space and sys_show_space functions, described in this table.
sys_get_rep_type({string <projdir>}, {string <system>})	string	Displays the repository type for a given system, as specified in the *.repinfo* file. You can use the output of this function as a parameter for the sys_create function. Possible values are "Sybase" and "MS Jet." If you make an error, it returns an empty string.
sys_has_dynamic_space({string <projdir>}, {string <system>})	boolean	Returns True if the system database can expand automatically as necessary. If the system cannot expand automatically, this function returns False. Systems for which this function returns True do not support the sys_expand_rep or sys_show_space function. The server that the system is on will not support repos_show_space. This function returns False for Sybase systems and True for Microsoft Jet systems.
sys_has_rep({string <projdir>}, {string <system>})	boolean	Verifies whether a system has a repository.
sys_list_current_users({string <projdir>}, {string <system>})	int	For Sybase only. Lists all users currently accessing the system. See also repos_can_list_current_users, described in Table 16 on page 3-68.

Table 18: System Functions (Continued)

Function	Return Type	Description
sys_list_users({string <projdir>}, {string <system>})	int	Lists all users who may gain access to the named system.
sys_load_rep({string <projdir>}, {string <system>}, {string <dumpdir>}, {optional_bool_enum <force>}, {optional_bool_enum <fast>}, {optional_bool_enum <loaduserinfo>})	int	For Sybase only. Loads a system's repository from a set of dump files created by the sys_dump_rep function or from the Desktop **Repository** menu (from the **Repository** menu, choose **Maintain Systems** > **Dump Current System Repository to Files**). Dump files are different from the system's flat files. Dump files are binary representations of the information in the repository. They contain information such as how StP objects are connected and what their attributes are. The system's flat files are human-readable diagram, table, and annotation files. All dump file information is also in the flat files, but the converse is not true; the flat files also contain information such as diagram layouts, which is not stored in the repository. A repository cannot be loaded while users are accessing the system. The target system does not have to be the system from which the dump files were dumped. In a secure system, the executor must be at least an owner. See also sys_dump_rep and sys_supports_rep_dump, described in this table.

Table 18: System Functions (Continued)

Function	Return Type	Description
sys_recover_rep({string <projdir>}, {string <system>}, {optional_bool_enum <force>}, {optional_bool_enum <fast>}) and sys_recover_rep({string <projdir>}, {string <system>}, {string <repositoryType>}, {int <size>}, {string <server>}, {int <filemode>}, {string <repname>}, {optional_bool_enum <force>}, {optional_bool_enum <fast>})	int	Regenerates a system repository from the system files. During regeneration, the system is locked, preventing all other system administration functions from being run concurrently. This is accomplished by placing a file, *.sysbusy*, at the system level, which is removed once the system has been rebuilt. sys_recover_rep calls sys_update, and all the same conditions hold with respect to potential writability of files. The executor must be a system administrator or system owner and must have write permissions on the files.
sys_show_space({string <projdir>}, {string <system>})	int	For Sybase, shows the size of the current system, available megabytes, and percent full. Display contains separate lines for the repository and transaction log. See also sys_has_dynamic_space, described in this table.
sys_supports_rep_dump({string <projdir>}, {string <system>})	boolean	Returns True if the system supports sys_dump_rep and sys_load_rep. This function returns True for Sybase and False for Microsoft Jet systems.
sys_trunc_hist({string <projdir>}, {string <system>}, {optional_bool_enum <keepusedonly>})	int	Whenever a user creates, updates, saves, or views a file, a file history object is created in the repository to note the action. Over time, these objects accumulate. sys_trunc_hist prunes all but the most recent file history object for each file. If you set <keepusedonly> to True, it prunes (per file) all but the most recent file history object for each user. In a secure system, the executor must be at least an owner.

Table 18: System Functions (Continued)

Function	Return Type	Description
sys_update [{string <projdir>}, {string <system>}, {optional_bool_enum <best_effort>}, {optional_bool_enum <force>}, {string <filenames>}, {string <filetypes>})	int	Adds new files to the repository and updates out-of-date files. sys_update updates the repository immediately, then updates the system files the next time they are loaded and saved in an editor. sys_update updates all system files by processing any pending renames. sys_update calls the **aupdate**, **tupdate**, and **dupdate** commands (which are described in *StP Administration*). The executor must have write access to all files to be processed. If an out-of-date file is under version control and is either checked out by another user or does not have proper file permissions, the command fails.

Parameters

Table 19: Sys* Function Parameters

Parameter	Description
best_effort	Lets sys_update update all the files it can and report any non-writeable files. Without this option, sys_update fails if it encounters a non-writeable file.
dumpdir	Used with sys_dump_load and sys_dump_rep to specify the target directory for repository dump files. If you do not specify the <dumpdir> parameter, the user will be prompted for a directory name.
dumpuserinfo	Dumps user information, such as the readers and writers of a system, with the system dump. The default is True.

Table 19: Sys* Function Parameters (Continued)

Parameter	Description
fast	Before loading files using sys_load_rep or sys_recover_rep, drops all indexes and triggers in the repository. If this parameter is false, performs a regular load, with indexes and triggers included. (Remember that regular loads can be slower than fast loads.)
filemode	The default file protection for table and diagram editor files. For Windows NT, the default is f (full control), as set with the Windows NT **cacls** command.
filenames	Used with sys_update to specify which files to update. Separate multiple file names with a space; do not separate each file name with a comma. You do not need to enclose the list in quotes. If you do not specify this parameter, all files are updated.
filetypes	Used with sys_update to specify which filetypes to update. Separate multiple file types with a space; do not separate each file type with a comma. You do not need to enclose the list in quotes. If you do not specify this parameter, all filetypes are updated.
force	Do not query user for confirmation. If you do not specify this parameter, StP prompts the user for confirmation of each action. If used with sys_update, files are updated whether or not they are identified as out-of-date. This is useful for updating files that exist in the system, when more recent versions have been manually copied over them.
keepusedonly	For each file, causes sys_trunc_hist to prune all but the most recent file history object for each user. Without this option, sys_trunc_hist prunes all but the single most recent file history object for each file. The default is False.
loaduserinfo	Loads user information, such as readers and writers of a system, with the system load. The default is False.

Table 19: Sys* Function Parameters (Continued)

Parameter	Description
newprojdir	Assigns this project name to the newly created system.
newsystem	Assigns this system name to the newly created system.
oldprojdir	The source project name.
oldsystem	The source system name.
password	The password to be assigned to a user or users; the default is "welcome" for Sybase. StP does not assign a default password for Microsoft Jet. By default, StP first checks the <server>_PASSWD environment variable for the password. If it is not set, it then checks the ToolInfo file for the *<server>_password* setting, as well as *<server>_admin_password*, *msjet_password*, and *msjet_admin_password* for the other password types. If you do not specify a password, StP prompts the user for one.
projdir	Uses the specified projdir directory as the current project directory (includes the drive designation in Windows NT). This projdir variable overrides the default directory specified by the ToolInfo variable, projdir. If you do not specify the <projdir> parameter, StP uses the default projdir directory.
readers	Specifies users who have read permission for the StP system. Separate the names with a space; do not separate each name with a comma.
repname	The name of the repository; the default is taken from the *system* ToolInfo variable.
repositorytype	The type of repository. Values are Sybase or MSJet.
repsize	The creation size of the repository in megabytes. It overrides the *syscreate_size* ToolInfo variable. The default is 6 MB.

Table 19: Sys* Function Parameters (Continued)

Parameter	Description
server	The name of the server; the default is the value of the DSQUERY environment variable. If DSQUERY is not set, StP assumes the repository is Microsoft Jet.
size	The amount of repository space in megabytes. For sys_expand_rep, the default is 2 MB. For sys_create and sys_recover_rep, the default is 6 MB.
system	Uses the named system as the current system name. This name overrides the default system name given by the *system* ToolInfo variable. The system name is appended to the project directory to specify the location of directories for the system.
users	The names of one or more users, which must be valid user names at the operating system level. Separate user names with a space; do not separate each name with a comma. You do not need to enclose the list in quotes.
writers	Specifies users who have write permission for the StP system. Separate the names with a space; do not separate each name with a comma.

Example

The following qrl script creates two new StP systems, one Sybase and the other Microsoft Jet, and then lists the users for those systems.

Figure 18: Sample Script: Creating Systems

```
int
main()
{
    /* First create new systems (negative return value
    indicates failure) */
    if (sys_create("c:/projects/", "newSybaseSystem",
    "Sybase") < 0)
    {
        message("Failed while creating system
        'newSybaseSystem'");
```

```
            return -1;
        }
        if (sys_create("c:/projects/", "newMSJetSystem",
    "MS Jet") < 0)
        {
            message("Failed while creating system
            'newMSJetSystem'");
            return -1;
        }
        /* Note:  This example uses the first form of sys_create,
        which uses the default values for the size, server,
        filemode and repname parameters. Check to see if the
        systems were created. */

        if (!sys_has_rep("c:/projects/", "newSybaseSystem"))
        {
            message("'newSybaseSystem' was not created
            correctly.");
            return -1;
        }
        if (!sys_has_rep("c:/projects/", "newMSJetSystem"))
        {
            message("'newMSJetSystem' was not created
            correctly.");
            return -1;
        }
        /* Check for available space */
        if (!sys_has_dynamic_space("c:/projects/",
        "newSybaseSystem"))
        {
            sys_show_space("c:/projects/", "newSybaseSystem");
        }
        if (!sys_has_dynamic_space("c:/projects/",
        "newMSJetSystem"))
        {
            /* We should not get to here */
            sys_show_space("c:/projects/", "newMSJetSystem");
        }
        /* List users for systems */
        if (sys_list_users("c:/projects/", "newSybaseSystem"))
        {
            return -1;
        }
        if (sys_list_users("c:/projects/", "newMSJetSystem"))
        {
```

```
        return -1;
    }
    return 0;
}
```

Setting Interface Elements for the Script Manager

You can use features in QRL to set interface elements for the Script Manager. The Script Manager is the main graphical user interface for the QRS.

Note: This section provides examples using **qrp** from the command line. Alternatively, you use the Script Manager to run scripts and set external variable values. For information, see Chapter 2, "The Script Manager."

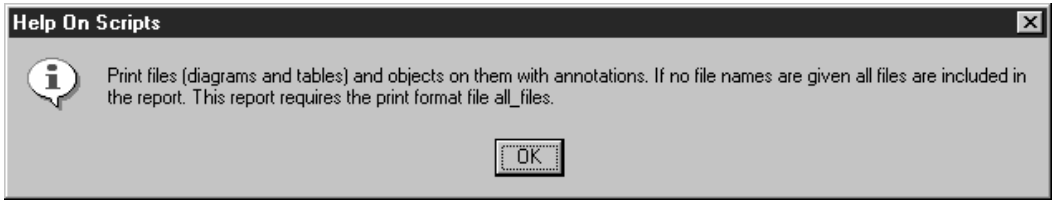
Help

QRL has a predefined <script_help> string variable. The value assigned to this variable becomes the help text the Script Manager displays for the script. For example:

```
script_help = "The help text you want to appear.";
```

As shown in Figure 19, help messages appear in the Script Manager Help window when you choose **On Script** from the **Help** menu.

Figure 19: Script Manager Help Window



External Variables

External variables can have values assigned to them at run time, either through the Script Manager or the command line. An external variable can be defined for a particular script, or it can be system-wide. The values of system-wide variables are shared by all the scripts in the system that refer to those variables.

External variables are declared within a script or an included file, and they can be declared with default values. Values provided in either the Script Manager or the **qrp** command line override the default values given in the script.

You can associate help text with an external variable for display in the Script Manager. For information on how you access external help through the Script Manager, see "Accessing Help on External Variables" on page 2-15.

To associate a help text with an external variable, assign <external_help> a value immediately following the external variable declaration:

```
external <QRL primitive type> <var_name> [ = <value>];
[external_help = <help text>;]
```

For example:

```
external string diagrams = "";
external_help = "Name(s) of Diagram(s) to Limit SQL DDL
   Generation.";
```

Use the same syntax to define system-wide external variables, but replace the keyword external with system_external.

The example in Figure 20 includes an external variable, <YEAR>. The script determines whether the value supplied for <YEAR> is a leap year and prints the results. Because QRL does not include a modulus operator, the script has a subfunction modulo that divides two integers and returns the remainder.

To run the script and provide a value for the external variable <YEAR>, at the command line type:

```
qrp -x YEAR 1988 year_example.qrl
```

where **-x** is the external command line option, YEAR is the name of the external variable, and 1988 is the value assigned to <YEAR>.

Note: To run scripts using the Script Manager, a graphical interface for running qrp, see Chapter 2, "The Script Manager."

Figure 20: Sample Script: Using an External Variable

```
// initialize YEAR
external int YEAR = 0;
void
main()
{
    int r_4, r_100, r_400;
    // Use modulo subfunction to determine the remainder when
    // you divide YEAR by 4, 100, and 400
    r_4 = modulo(YEAR, 4);
    r_100 = modulo(YEAR, 100);
    r_400 = modulo(YEAR, 400);
    if ((r_4 == 0 && r_100 !=0) || r_400 == 0 )
       print_line(YEAR + " is a leap year.");
    else
       print_line(YEAR + " is not a leap year.");
}
int modulo (int number, int modulus)
{
    int division = number/modulus; // may be rounded
    int remainder = number - (division * modulus);
    // The remainder will be in the
    // range -(modulus) < remainder < modulus.
    // We always want the remainder to be positive,
    // so if it's not, add the modulus to it.
     if (remainder < 0)
       remainder = remainder + modulus;
    return remainder;
}
```

The output from this script as run with the above command line is:

```
1988 is a leap year.
```

Note that you can use a null string to initialize an external variable of the type string:

```
external string S_VAR = "";
```

A value supplied for a string external variable in the command line must be enclosed by quotes:

```
qrp -x S_VAR "abc" filename.qrl
```

Extensible Formatting

QRL provides built-in functions for selecting formatting options from the target publishing system catalogs. Because RTF (Microsoft Word), HTML, and FrameMaker have extensible formatting catalogs, this is a powerful mechanism. For a description of these functions, see Chapter 5, "Creating Documents."

Troubleshooting

This section describes common mistakes to avoid in writing scripts.

Common Programming Mistakes

Some of the more common programming mistakes are:

- Misplacing a semi-colon
- Confusing the = operator with the == operator

Misplacing a Semi-Colon

An example of misplacing a semi-colon is:

```
for(i = 0; i < 10; i = i + 1);
   print_line("This statement will never be executed");
```

Placing a semi-colon immediately after this for loop results in an error. This same type of mistake is common in `if` statements and while loops.

Confusing Operators

An example of confusing the = operator with the == operator is:

```
void main()
{
    int i = 34;
    if( i = 1 )
       print_line("Out of range");
}
```

Using the assignment operator (=) in the if statement rather than the “is equal to” operator (==) causes the script to work improperly. The if statement assigns the value of 1 to `i` instead of testing to see if i == 1. This causes the if statement to evaluate to True (since any non-zero value is considered True), so that the print_line statement is always executed.

Loops and Enumerations

The following example shows a loop over an enumeration that generates and error: “Operation failed with status: Illegal Null Operand”:

```
enum Anytown {Smith, Jones};
void main ( )
{
    Anytown resident;
    for (resident = enum_first(“Anytown”);
         resident <= enum_last(“Anytown”);
         resident = enum_next(resident)) {
        print_line(resident);
    }
}
```

The error occurs because after the second (and what should be the last) time through the loop, the resident is set to `enum_next (Jones)` which evaluates to NULL, and then the loop control expression is evaluated, comparing resident (which is NULL) to enum_last (“Anytown”).

The correct code is:

```
enum Anytown {Smith, Jones};
void main ( )
{
```

```
    Anytown resident;
    for (resident = enum_first(“Anytown”);
         resident != NULL;
         resident = enum_next(resident)) {
        print_line(resident);
    }
}
```

Similarly, a loop running backwards cannot be:

```
enum Anytown {Smith, Jones};
void main ( )
{
    Anytown resident;
    for (resident = enum_last(“Anytown”);
         resident >= enum_first(“Anytown”);
         resident = enum_prev(resident)) {
        print_line(resident);
    }
}
```

The correct code is:

```
enum Anytown {Smith, Jones};
void main ( )
{
    Anytown resident;
    for (resident = enum_last(“Anytown”);
         resident != NULL;
         resident = enum_prev(resident)) {
        print_line(resident);
    }
}
```

Declaring Local Variables

Local variables should not be declared within a loop. Unlike C, when you declare a variable in QRL in a block that is not a function, the variable is not destroyed when you leave the block. The variable is usable for the rest of the function, regardless of which block it is declared in. This can have unexpected results if you declare a local variable inside a loop. For example:

```
void QRL_foo_for(int condition)
{
    int ix, count = 10;
    int local_variable;
    for (ix = 0; ix < count; ix = ix + 1)
     {
        if (condition)
        // In the second loop, QRL will try to redeclare
        // local_variable,
        // but it still exists, so you will get an error.
        int local_variable;
        local_variable = subfunc();
        ...
     }
        else
        ...
}
```

To avoid the error in QRL_foo_for, declare the local variable outside the loop:

```
void QRL_foo_for(int condition)
{
    int ix, count = 10;
    int local_variable;    // ********* HERE *********
    for (ix = 0; ix < count; ix = ix + 1)
    {
        if (condition)
    {
        // local_variable will be reset but not redeclared
        local_variable = subfunc();
        ...
    }
        else
        ...
     }
    }
```

Passing list, set, and graph Values

List, set, and graph objects are passed by reference. In other words, if you pass a list, set, or graph into a function and make changes to the formal parameter within the function (by using list_clear, list_set, and so on), the actual parameter (the original list, set, or graph) is modified as well.

However, assigning a list, set, graph, or single element to a variable makes a copy of the original. Subsequently changing the value of the variable does not cause a corresponding change in the original list, set, graph, or element. Consequently, to change the value of an element in a list that has been assigned to a variable, for example, you must explicitly update the list by using the list_set function.

This example illustrates how and where values are assigned to elements in a list called numbers:

Figure 21: Sample Script: Assigning Values

```
// Wrapper
void main()
{
   func();
}
   void func()
   {
       list numbers = list_create("int", 0);
       int ix, count = 0;
// ints and other scalars are passed by value; count is
// not reset.
       int_init(count);
       print_line("count has the value " + count);
       count = 10;
       for (ix = 0; ix < count; ix = ix + 1)
           list_append(numbers, ix);
// at this point we have a list of numbers 0-9
       subfunc(list_get(numbers, 0), 3);
// numbers[0] is still 0 here.
       print_element(numbers, 0);
       subfunc2(numbers, 0, 3);
// numbers[0] is 3 here.
       print_element(numbers, 0);
       subfunc3(numbers, 0, 5);
// numbers[0] is still 3 here.
       print_element(numbers, 0);
       list_set(numbers, 0, subfunc4(5));
// numbers[0] is 5 here.
       print_element(numbers, 0);
   }
   void int_init(int number)
   {
```

```
        number = 10;
    }
    void subfunc(int element, int value)
    {
        element = value;
    }
    void subfunc2(list digits, int index, int value)
    {
        list_set(digits, index, value);
    }
    void subfunc3(list digits, int index, int value)
    {
        int element = list_get(digits, index);
        element = value;
// If we used the next line, the list would change:
// list_set(digits, index, element);
     }
    int subfunc4(int value)
    {
        return value;
    }
    void print_element(list the_list, int index)
    {
        print_line("Index: " + index + " has the value " +
        list_get(the_list, index));
    }
```

The output of this script is:

```
count has the value 0
Index: 0 has the value 0
Index: 0 has the value 3
Index: 0 has the value 3
Index: 0 has the value 5
```

QRL Debugging Features

This section describes how to use special **qrp** options and built-in functions to trace the execution of QRL scripts and display debugging messages during a debugging session.

Note: QRL tracing is available from the command line only.

QRL includes the following debugging facilities:

- Predefined trace symbols
- Built-in trace functions
- **qrp** command trace options

Tracing is the ability to log the activities of a QRL script as it executes, which provides information on the sequence of events. By tracing the order in which functions are called, you can follow the path of the executing program. If the program is having trouble, this helps you to locate the problem.

You can trace a specific function, all user-defined functions, all built-in functions, all function returns, or any combination of these during a single debugging session. This requires modifications to the QRL script being debugged, but debugging messages are not displayed unless requested, either by a call to a built-in trace function, or by using a **qrp** command line option when you run the script.

This section first presents descriptive lists of the trace features—predefined symbols, built-in functions, and **qrp** command options—then provides a discussion of several examples using these trace features.

Predefined Trace Symbols

You can use the QRL built-in trace functions or **qrp** trace options to report on a single function. QRL also provides predefined symbols that you can use as arguments to report on functions by type or even function returns. One predefined symbol enables the printing of timestamps in front of trace messages, which profiles the execution to determine where time is being spent.

Table 20 lists the predefined symbols to use for debugging QRL scripts.

Table 20: Predefined Trace Symbols

Predefined Symbol	Meaning
calls	All user-defined functions
builtins	All QRL built-in functions
returns	All function return values
all	All calls, builtins, and returns
none	No calls, builtins, or returns
timestamps	Puts a timestamp in front of each trace message indicating the elapsed time since **qrp** began execution

Built-In Trace Functions

QRL's built-in trace functions display debugging messages and turn various tracing features on and off. These are activated by the **qrp** command options listed in Table 1 on page 3-5.

Table 21 lists the built-in trace functions.

Table 21: Built-In QRL Trace Functions

Function	Return Type	Description
trace(string <symbol>)	string	Enables tracing of functions and all trace_print() messages for <symbol>. <symbol>can be a specific function or one of the predefined symbols listed in Table 20 on page 3-93.

Table 21: Built-In QRL Trace Functions (Continued)

Function	Return Type	Description
untrace(string <symbol>)	string	Disables the tracing of functions and the display of trace_print() messages for <symbol>. <symbol> can be a specific function or the predefined symbol “calls,” “builtins,” “returns,” or “all.”
trace_print(string <symbol>, string <any_string>)	string	Displays <any_string> if tracing is enabled for <symbol>.
trace_file(string <output_file>)	string	Directs trace output from the default (stderr) to <output_file>.

For discussion and examples using the built-in trace functions, see “Using the QRL Debugging Tools” on page 3-96.

qrp Command Trace Options

The **qrp** command trace options enable QRL’s debugging features. Table 1 lists these options.

Table 22: qrp Trace Command Options

Option	Argument	Description
-trace	calls	Turns tracing on for all calls to user-defined functions.
	<function_name>	Turns tracing on for all calls to <function_name>.
	returns	Turns tracing on for all function returns, displaying the return value of each function called.
	builtins	Turns tracing on for all calls to built-in QRL functions, such as the list and string manipulation functions.
	timestamps	Turns profiling on in conjunction with any other trace output requested. Each trace message is preceded by a timestamp indicating the elapsed time from the moment **qrp** began execution.
	all	Turns tracing on for all trace options except “timestamps.”
-trace_file	<output_file>	Redirects trace output from the default (stderr) to <output_file>.

Using the QRL Debugging Tools

This section presents several examples of typical debugging tasks using the QRL tools. The script in Figure 22 uses several built-in trace functions.

Note: QRL tracing is available from the command line only.

Figure 22: Sample Script: Using QRL Debugging Tools

```
void
main()
{
   int main_int_1;
   print(function_D(5, 14, "Hello World!") + "\n");
}

void function_A(string A_string_1)
{
   print(A_string_1 + "\n");
}

void function_B(string B_string_1, int B_int_1)
{
   print("Here I am!" + "\n");
}

void function_C(int C_int_1, int C_int_2)
{
   trace_print("C", "I'm in function C now");
   untrace("C");
   trace_print("C", "This shouldn't print out");
   function_B("Hey there!", C_int_1);
}

string function_D(int D_int_1, int D_int_2,
      string D_string_1)
{
   trace("function_A");
   function_C(12, D_int_1);
   function_A(D_string_1);
   return("Goodbye, cruel world... *sob*.");
}
```

trace_print and untrace stateme

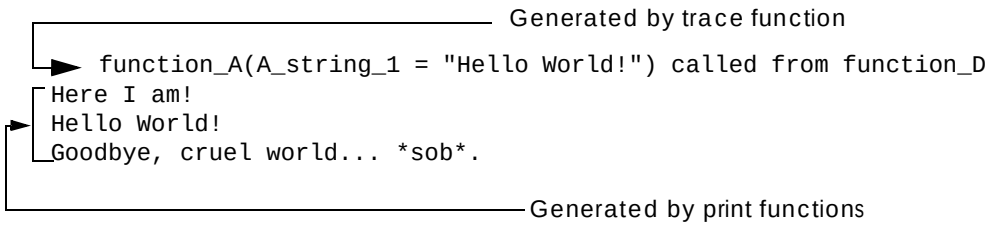
trace and return statements

To run this script without command line options, enter:

```
qrp debug.qrl
```

where *debug.qrl* is the name of the script. The output of this command displays the messages shown in Figure 23.

Figure 23: Sample Trace Output: Without Command Options



The first line of output is the only one that does not result from ordinary print() statements. This line is generated by the `trace("function_A");` statement.

The other trace functions in the script are not activated until you turn them on with a **-trace** command line option.

For a discussion on activating these functions, see "Tracing Specific Elements" on page 3-98.

Redirecting Trace Output

Unless redirected, both ordinary script output (from print statements, for example) and debugging messages are displayed on stderr. To redirect ordinary output (from print statements, for example), use the **-o** <output_file> command option. To redirect debugging messages, use either the command line option **-trace_file** <output_file> or the built-in function trace_file(<output_file>).

Redirecting output from within the program (using trace_file) overrides any command line specification. For example, to redirect debugging output to the *msgs.txt* file using command line options, type:

```
qrp -trace_file msgs.txt debug.qrl
```

The screen (stdout) output is:

```
Here I am!
Hello World!
Goodbye, cruel world... *sob*.
```

The contents of the *msgs.txt* file are:

```
function_A(A_string_1 = "Hello World!") called from
          function_D
```

To redirect debugging output from the code, add a call to the trace_file function. For example:

```
void function_C(int C_int_1, int C_int_2)
{
   trace_print("C", "I'm in function C now");
   untrace("C");
   trace_print("C", "This shouldn't print out");
   function_B("Hey there!", C_int_1);
   trace_file("trace_msgs.txt");
}
```

Call to trace_file function

Issue the same command as before:

```
qrp -trace_file msgs.txt debug.qrl
```

This time, the **-trace_file** *msgs.txt* option is ignored (the file is not created). Instead, the trace_file function creates the *trace_msgs.txt* file with the contents:

```
function_A(A_string_1 = "Hello World!") called from
          function_D
```

Note: Both the **-trace_file** option and the trace_file() function append messages to the file, so that when you rerun the script, new messages are added to the old ones, rather than writing over them.

Tracing Specific Elements

You use the predefined trace symbols from Table 20 on page 3-93 to specify elements to trace. You use these either as arguments to the **-trace**

option of the **qrp** command or as arguments to the built-in trace functions.

For example, in *debug.qrl* (see Figure 22 on page 3-96), there are several <function_name> arguments to trace functions:

- From function_C:

```
trace_print("C", "I'm in function C now");
untrace("C");
trace_print("C", "This shouldn't print out");
```

- From function_D:

```
trace("function_A");
```

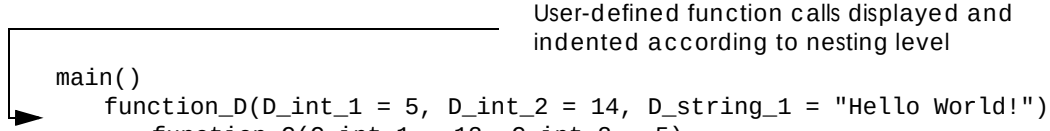
The next few sections illustrate the results of activating tracing for various elements in *debug.qrl* (Figure 22 on page 3-96).

Tracing All User-Defined Functions

To trace all user-defined functions, type:

```
qrp -trace calls -trace_file msgs.txt debug.qrl
```

The contents of *msgs.txt* are:




```
main()
   function_D(D_int_1 = 5, D_int_2 = 14, D_string_1 = "Hello World!")
      function_C(C_int_1 = 12, C_int_2 = 5)
         function_B(B_string_1 = "Hey there!", B_int_1 = 12)
      function_A(A_string_1 = "Hello World!") called from function_D
```

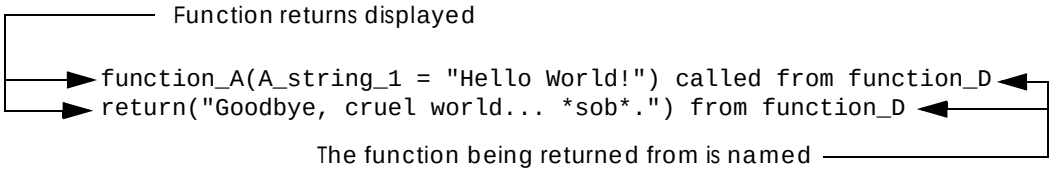
Each user-defined function call is displayed with the value of each argument at the time of the call. Functions are indented according to nesting level (for example, function_B is indented under function_C because function_C called function_B).

Tracing All Function Returns

To trace all function returns, type:

```
qrp -trace returns -trace_file msgs.txt debug.qrl
```

The contents of *msgs.txt* are:



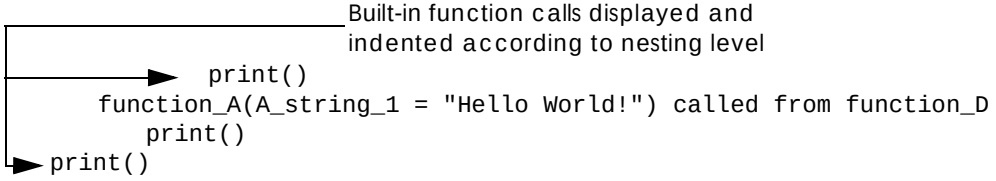
Each function return is displayed, along with the name of the function the return is being returned from.

Tracing All Built-In Functions

To trace all built-in functions, type:

```
qrp -trace builtins -trace_file msgs.txt debug.qrl
```

The contents of *msgs.txt* are:



This output shows the trace statement and the three print() statements that generate the three lines of program output. These are indented according to nesting level.

Tracing All Elements

To trace all user-defined and built-in functions, as well as function returns (that is, to combine the three previous examples), type:

```
qrp -trace all -trace_file msgs.txt debug.qrl
```

The contents of *msgs.txt* are:

```
main()
   function_D(D_int_1 = 5, D_int_2 = 14, D_string_1 = "Hello
               World!")
      function_C(C_int_1 = 12, C_int_2 = 5)
         function_B(B_string_1 = "Hey there!", B_int_1 =
               12)
            print()
      function_A(A_string_1 = "Hello World!") called from
               function_D
         print()
      return("Goodbye, cruel world... *sob*.") from
               function_D
   print()
```

This output shows everything in the three previous examples combined.

Turning On trace_print Functions

To see the debugging messages displayed by the trace_print() function for a particular function, you must explicitly request tracing of that function. For example, type:

```
qrp -trace C -trace_file msgs.txt debug.qrl
```

The contents of *msgs.txt* are:

```
   [Trace C] I'm in function C now
function_A(A_string_1 = "Hello World!") called from
         function_D
```

This output displays the message generated by the line:

```
trace_print("C", "I'm in function C now");
```

In the *debug.qrl* script (review Figure 22 on page 3-96), this is followed by these lines:

```
untrace("C");
trace_print("C", "This shouldn't print out");
```

Note that the untrace function turns off tracing, so that calls to trace_print after it are ignored.

Profiling the Script

You use the "timestamp" argument to determine how much time is being spent at each function call and return. To determine how long things take executing the *debug.qrl* script, type this command:

```
qrp -trace all -trace timestamps -trace_file msgs.txt
      debug.qrl
```

The contents of the *msgs.txt* file shows the elapsed time in milliseconds at each function call and return.

```
1.520:main()
1.612:  function_D(D_int_1 = 5, D_int_2 = 14, D_string_1 = "Hello World!")
1.664:    function_C(C_int_1 = 12, C_int_2 = 5)
1.670:      function_B(B_string_1 = "Hey there!", B_int_1 = 12)
1.672:        print()
2.628:    function_A(A_string_1 = "Hello World!") called from function_D
2.630:      print()
2.632:    return("Goodbye, cruel world... *sob*.") from function_D
2.634:  print()
```

4 The OMS Query Language and QRL

This chapter describes how to use Object Management System (OMS) queries in QRL scripts to access objects in the repository. The sections in this chapter are:

The OMS defines a set of abstract types used to model persistent data maintained in the repository. QRL includes a set of corresponding PDM data types that allow access to repository objects. From a script you can use PDM types in OMS queries to:

- Return information about objects in the repository
- Display objects in the Repository Browser
- Create a list or set of selected objects

For more information on writing QRL scripts, see Chapter 3, "The StP Query and Reporting Language." PDM types are listed in "PDM Types" on page 3-13. To learn how to construct an OMS query, see *Object Management System*. For more information on the Repository Browser, see *Fundamentals of StP* or the documentation specific to your StP product.

OMS Interface Functions

QRL has several built-in functions that allow you to perform interface operations with the OMS. You can use these functions in a QRL script to return information about repository objects. Table 1 lists the OMS interface functions available in QRL.

Table 1: OMS Interface Functions

Function	Return Type	Description
for_each_in_select (string <oms_query>, <PDM_type> <query_obj>)	<PDM_ type>	Retrieves repository objects specified by <oms_query>.
find_by_query (string <oms_query>)	<PDM_ type>	Returns the first object matching the query or NULL if no match.
selection_count (string <oms_query>)	int	Returns the number of objects selected by <oms_query>.
app_type_print_string (<PDM_type> <object>)	string	Returns a printable representation of the <object>'s application type.
id_list_create (string <oms_query>, string <id_list_name>)	void	Creates a list of the IDs of currently selected objects.
id_list_free (string <id_list_name>)	void	Deletes list that was created to contain object IDs.
list_select(string <oms_query>)	list	Executes <oms_query> and returns a list containing the results. The list type corresponds to the query.

Table 1: OMS Interface Functions (Continued)

Function	Return Type	Description
set_select(string <oms_query>)	set	Executes <oms_query> and returns a set containing the result. The returned set is of the type corresponding to the query.
to_oms_string (string <the_string>)	string	Escapes OMS special characters (that is, apostrophes) in <the_string>; applied to QRL strings that are to be used as string literals in OMS queries. Returns <the_string> with the special characters escaped.

These functions are described in the following sections.

Another built-in function used in QRL scripts that include OMS queries is browse, which invokes the Repository Browser and displays objects specified in a query. For more information on browse, see "Browsing the Repository" on page 4-18.

Using the for_each_in_select Function

You can use the for_each_in_select function in a QRL script to retrieve PDM type objects from the repository. This function is a hybrid of a function call and a `for` loop—essentially, a parameterized `for` loop.

The for_each_in_select loop statement has the following form:

```
for_each_in_select(string <oms_query>, <PDM_type>
    <query_obj>)
{
    loop_body
}
```

The effect of executing this statement is:

1. The <oms_query> is executed, retrieving a set of PDM type objects from the repository.
2. The loop_body is executed once for each object in the retrieval.
3. The <query_obj> has the value of an object returned by the query each time through the loop.

A simple selection is used to access *all* objects of a given type; that is, no selection criteria are applied to the query. This type of selection takes the form:

```
for_each_in_select("<PDM_type>", pdm_var)
```

For example, the following expression retrieves all nodes in the repository:

```
for_each_in_select("node", node_var)
```

This example script retrieves all the nodes on the system and prints the name and type of each node.

```
void
main()
{
   node node_var;
   for_each_in_select("node", node_var)
      print_line("The node \"" + node_var.name+ "\" is" +
                  " of the type " + node_var.type);

}
```

Note: The print_line statement uses node_var.name and node_var.type to reference the “name” and “type” attributes of the current node, node_var.

Depending on what you have on your system, the output of this script resembles the following:

```
The node "begin class" is of the type UMLClass
The node "end class" is of the type UMLClass
.
.
.
```

Applying Conditions to Selections

Selections can apply conditions to a retrieval that describe relationships between objects and values of attributes. The syntax is:

```
for_each_in_select
    ("<PDM_type> [<OMS query language expression>]",
       <pdm_var>)
```

For example, this statement retrieves all nodes in the system of type UmlClass:

```
for_each_in_select("node[type == UmlClass]")
```

Note that `"type ==" ` can be omitted from this query without changing the result:

```
for_each_in_select("node[UmlClass]")
```

Similarly, the query:

```
for_each_in_select("node[id == 3803]");
```

can be expressed as:

```
for_each_in_select("node[3803]");
```

Any legal OMS query can be used in such a selection. With the exception of substitution variables (discussed in "Using Variable Substitution" on page 4-12), QRL passes the query string as is to the OMS; that is, QRL does not process the query.

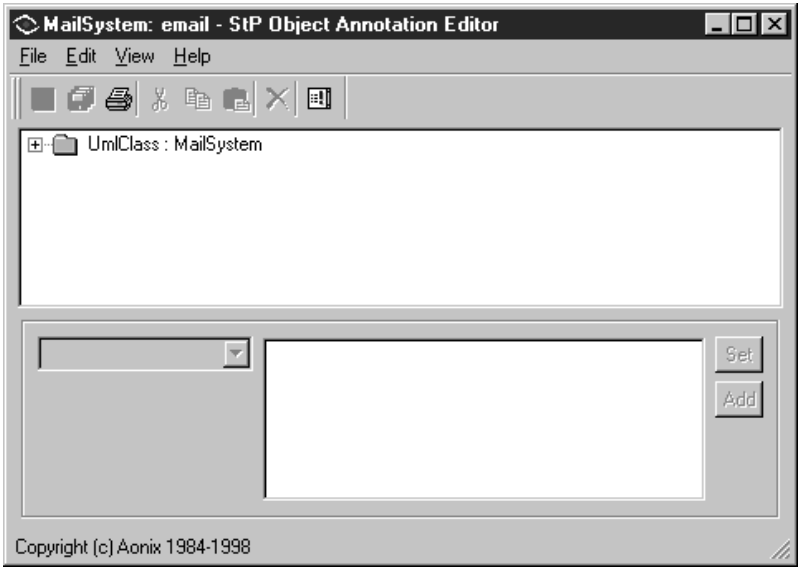
This example script retrieves a Class node named *MailSystem* from the repository. It then uses the node's id attribute in the **stpem** command line (which is specified as an argument to the system function) to open the Object Annotation Editor for *MailSystem*.

```
void
main()
{
    node node_var;
    for_each_in_select("node[UmlClass &&
        name == 'MailSystem']", node_var)
    // Use the stpem command to open the Object Annotation
    // Editor for the UML class "MailSystem"
    system("stpem -ed oae -C \"AnnotEdit " +
```

```
        node_var.id + "\"");
}
```

The result of running this script is that the Object Annotation Editor opens, displaying annotation information for *MailSystem*.

Figure 1: The Object Annotation Editor



For more information about using the system function, see Chapter 3, "The StP Query and Reporting Language."

Sorting Retrievals

You can use the "sort by" query string argument to sort objects based on a specified attribute. For example, the following script retrieves all files except for ObjectAnnotation and UmlClassDiagram files, sorts the retrieved files by the attribute name, and prints the name and type of each file:

```
void
main()
{
   file f_var;
```

```
    for_each_in_select("file[!ObjectAnnotation &&" +
        " !UmlClassDiagram] sort by name", f_var)
            print_line(f_var.name + " " + f_var.type);
}
```

Using the find_by_query Function

The find_by_query function returns the first object matching the query string, or NULL if there is no match. If you are searching for a unique object by key, find_by_query is the most efficient way to retrieve it from the repository.

The syntax is:

```
find_by_query(string <oms_query>)
```

This example script searches for a note that has the id attribute 73801. If it finds a match, it prints the node. Otherwise, it returns NULL.

```
void
main()
{
    int id = 73801;
    print(find_by_query("note[id == "+id+"]"));
}
```

Using the selection_count Function

The selection_count function returns the number of objects found by a query.

The syntax is:

```
selection_count(string <oms_query>)
```

The following example searches for node objects of the type UmlClass that have Requirements notes. If no objects matching the query are found (that is, if count == 0), the script prints the message "No objects with Requirements notes found." Otherwise, the script prints the number of objects found (that is, the value returned by `selection_count(query)`).

```
void
main()
```

```
{
   string query;
   int count;
   query = "node[UmlClass && notes[Requirement]]";
   count = selection_count(query);
   if(count == 0)
      print("No objects with Requirements notes found.");
   else
      print("Number of objects found that have " +
            "Requirements notes: " + count);
}
```

Using the app_type_print_string Function

The app_type_print_string function returns a printable version of an object's application type.

The syntax is:

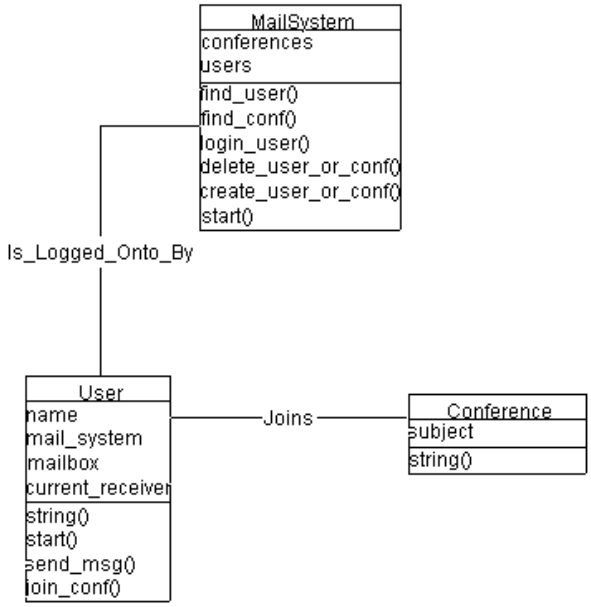
```
app_type_print_string(<PDM_type> <object>)
```

This example script prints the name (if any) and application type for all of the nodes on the system.

```
void
main()
{
   node current_node;
   string app_type;
   for_each_in_select("node", current_node)
   {
      app_type = app_type_print_string(current_node);
      print_line("Node \"" + current_node.name +
            "\" has the application type " + app_type);
   }
}
```

For example, consider that you had only the following UML diagram on your system:

Figure 2: UML Diagram



The output of the script would include:

```
Node "MailSystem" has the application type UmlClass
Node "User" has the application type UmlClass
Node "Conference" has the application type UmlClass
```

Using the id_list_create and id_list_free Functions

ID lists are a way of limiting queries to a smaller set of objects than the entire repository. They are useful when a script performs repeated queries that share common constraints, such as a checking program that always selects objects contained in a particular diagram.

Using the id_list_create function creates an ID list containing the IDs of all of the objects in the current selection. This allows you to perform operations on the objects without fetching them multiple times from the

repository. When you are finished using the list of IDs, you use the id_list_free function to delete the list.

The syntax is:

```
// to create an id list:
id_list_create(string <oms_query>, string <id_list_name>)
// to free an id list:
id_list_free(string <id_list_name>)
```

In the following example, an ID list "node_list" is created that contains all nodes of the type UmlClass on the system. This ID list is used for all subsequent queries. Finally, the ID list is freed.

```
void
main()
{
   string query;
   node current_node;
   // Save the list of IDs so that the repository doesn't
   // have to fetch them more than once
   query = "node[UmlClass]";
   id_list_create(query, "node_list");
   // Select all elements in list; print their types
   // and names
   query = "node[node_list]";
   print_line("Complete list of nodes in node_list: \n");
   for_each_in_select(query, current_node)
      print_line(current_node.type + " " +
         current_node.name);
      print("\n");
   // Select all nodes whose names start with "p"
   query = "node[node_list && name$'p*']";
   print_line("All nodes whose name starts with \"p\": \n");
   for_each_in_select(query, current_node)
      print_line(current_node.type + " " +
         current_node.name);
      print("\n");
   // Select all nodes that are connected to another
   // symbol
   query = "node[node_list && (in_links || out_links)]";
   print_line("All nodes that have in or out links: \n");
   for_each_in_select(query, current_node)
      print_line(current_node.type + " " +
         current_node.name);
```

```
        print("\n");
    // Select all nodes that have a base note with an
    // author item
    query = "node[node_list && " +
           "notes[(GenericObject && items[author])]]";
    print_line("All nodes that have a base note " +
           "with an author item: \n");
    for_each_in_select(query, current_node)
        print_line(current_node.type + " " +
            current_node.name);
        print("\n");
    // Free ID list
    id_list_free("node_list");
}
```

Using the list_select and set_select Functions

The list_select and set_select functions enable you to execute an OMS query and return a list or set, respectively. The list or set returned has the type specified in the OMS query.

This example script uses the list_select function to create a list containing nodes of the type UmlClass that have out links. Attempts to add elements of a different data type to the list would fail.

```
void
main()
{
    list my_list;
    string query;
    query = "node[UmlClass && out_links]";
    my_list = list_select(query);
    print(my_list);
}
```

This example can also be modified to create a set by substituting set_select for list_select. For more information on using lists and sets, see Chapter 3, “The StP Query and Reporting Language.”

Using the to_oms_string Function

The to_oms_string function takes a string literal as an argument, checks it for apostrophes (OMS special characters), and inserts a backslash (\) before the apostrophes to "escape" them. It returns the string literal with the special characters escaped.

The to_oms_string function is applied to QRL strings that are to be used as string literals in OMS queries. This simple example illustrates how the to_oms_string function works:

```
void
main()
{
   string str1, str2;
   str1 = "What's in a name?";
   str2 = to_oms_string(str1);
   print_line(str2);
}
```

The output from this script is:

```
What\'s in a name?
```

Using Variable Substitution

The preceding examples in this chapter have used string literals (such as "UmlClass") as the operand to compare with the value of the type attribute. Normally, QRL passes query strings to the OMS without processing them. However, you can use variable substitution in queries to signal QRL to process the query string (that is, to replace the variable name with its value) before passing the query to OMS. This is particularly useful in constructing nested queries (see "Using Nested Queries" on page 4-15).

The syntax for variable substitution in a query is:

```
${<variable to be substituted>}
```

Example

In the following example, the external variable FILE is used as a substitution variable in an OMS query. The script prints information about links on the diagram *proj_diag*, which is declared as the value of FILE. The benefit of using a substitution variable in this script is that you can supply a different diagram name as a value for FILE, and the new name is substituted in the query.

```
external string FILE = "proj_diag";
void
main()
{
    link link_var;
    int i = 1;
    // The external variable FILE is used as a substitution
    // variable in this query
    for_each_in_select("link[link_refs[file" +
        " [name == '${FILE}']]]", link_var)
    {
        print_line("\nLink #" + i + ":\n");
        print_line("Link type: " + link_var.type);
        print_line("Link name: " + link_var.name);
        print_line("Link ID: " + link_var.id);
        print_line("from_node ID: " + link_var.from_node_id);
        print_line("to_node ID: " + link_var.to_node_id);
        i = i + 1;
    }
}
```

The output of this script would resemble the following:

```
Link #1:
Link type: UmlTransition
Link name: now
Link ID: 77309
from_node ID: 77303
to_node ID: 77305
Link #2:
Link type: UmlTransition
Link name: before
Link ID: 77510
from_node ID: 77503
to_node ID: 77505
. . .
```

Variable Substitution and Scope

When using variable substitutions, keep in mind that QRL looks for the value of a variable in the variable's current scope. When a substitution variable is referenced outside of the scope in which it was defined, QRL is unable to replace it with the appropriate value. This can lead to problems if a function tries to use a query that contains substitution variables that were defined in a different scope.

For example, the following script uses a locally defined substitution variable in a query in main(), and then passes the query as an argument to sub_func. This results in an error, because from sub_func, QRL is unable to determine a value for the substitution variable. Three possible solutions to this problem are embedded in the script's comments; using any one of them causes the script to run without error.

```
// This script will not work properly as is. You can use
// any ONE of the proposed solutions to get the script to
// run without error.
// SOLUTION 1: Declare ID as an external variable:
// external int ID = 65000;
// SOLUTION 2: Simply declare ID globally:
// int ID = 65000;
void
main()
{
   node node_var;
   int ID = 65000;
   // Passing this query to sub_func will result in an error
   string query = "node[id < ${ID}]";
   // SOLUTION 3: Use this query instead:
   // string query = "node[id < "+ID+"]";
   for_each_in_select(query, node_var)
      print_line(node_var.id);
   sub_func(query);
}
void
sub_func(string func_query)
{
   node func_node;
   // The query here will be passed as a literal string;
   // the substitution will not be made
   for_each_in_select(func_query, func_node)
```

```
        print_line(func_node.name);
}
```

Using Nested Queries

QRL allows nested queries, where the value of an attribute retrieved in the outer selection can be used as an operand in the query that is applied to the inner selection.

Generally, it is best to avoid nesting OMS queries, but if objects share a name association, nested queries may be unavoidable.

Nested queries are accomplished by using variable substitution in a query to signal QRL that it must process the query (that is, replace substitution variables with their values) before passing the query to the OMS. For more discussion of variable substitution, see "Using Variable Substitution" on page 4-12.

In the following example script, the outer loop uses a variable "node_var" to store the value of the current node object, and then the value of node_var.id is compared to the obj_id attribute of the note type in the inner select query. Using node_var.id as a substitution variable in the inner loop query enables QRL to supply the appropriate value for the variable each time through the loop.

The script prints the names of all of the nodes on the system and their associated notes.

```
void
main()
{
    node node_var;
    note note_var;
    for_each_in_select("node", node_var)
    {
        print_line("\nNode: " + node_var.name);
        for_each_in_select("note[obj_id == " +
            "${node_var.id}]", note_var)
                print_line("Note: " + note_var.name);
    }
}
```

The next example (Figure 3) expands on this theme and creates a structure containing information about a node, which includes a node object and a list of the notes associated with the node. The script then uses a nested query to retrieve the notes for each object and appends each note to a list. Finally, the script prints each object and the notes associated with it.

Figure 3: Sample Script: Nested Queries

```
struct node_info
{
   node object;
   list note_list;
};
void
main()
{
   node_info node_struct;
   note note_var;
   node_struct.note_list = list_create("note", 0);
   for_each_in_select("node[UmlClass]" +
      "sort by name", node_struct.object)
   {
      print_line("\nObject:");
      print_line(node_struct.object);
      // The ${node_struct.object.id} substitution
      // variable in this inner for_each_in_select
      // loop indicates to QRL to replace
      // the variable with the value of the id
      // attribute before passing the query to OMS
      for_each_in_select("note[obj_id == " +
         "${node_struct.object.id}]", note_var)
      {
         print_line("\nNote: ");
         list_append(node_struct.note_list, note_var);
         print_line(note_var);
      }
   }
}
```

This script uses the print_line function to print a list of objects and their associated notes. The output of this script would resemble the following:

```
Object:
{
id = 81105
name = Mission
type = UmlClass
scope_node_id = 0
sig =
annot_file_id = 81503
}
Note:
{
id = 81603
obj_id = 81105
name = Geordi
type = Resource
file_id = 81503
desc = Chief Engineer
}
Note:
{
id = 81605
obj_id = 81105
name = Lieutenant Commander Data
type = Resource
file_id = 81503
desc = android
}
Note:
{
id = 81606
obj_id = 81105
name = Worf
type = Resource
file_id = 81503
desc = Security Chief
}
Note:
{
id = 81607
obj_id = 81105
name = warp drive
type = Tool
file_id = 81503
desc =
}
```

```
Note:
{
id = 81608
obj_id = 81105
name = tractor beam
type = Tool
file_id = 81503
desc =
}
Note:
{
id = 81610
obj_id = 81105
name = anti-matter
type = Tool
file_id = 81503
desc =
}
. . .
```

An example of how this script can be modified to display its output to the Repository Browser is given in "Browsing the Repository,"which follows.

Browsing the Repository

The browse function is used in a QRS script to invoke the Repository Browser and display the objects specified in a query. It does not specify *how* to display PDM objects in the Browser. The way PDM objects are formatted once they are in the Browser is specified in a rules file.

The Browser uses the generic table editor to display PDM objects. The formatting specifications of PDM objects in the table editor are:

- Each row contains one PDM object.
- Each column contains an identifying attribute of the PDM object.
- Each section contains one type of PDM object.

  That is, PDM objects of the same type are grouped into separate sections of the table.

The browse function takes any one of the following as an argument:

- String

  A string passed to the browse function is interpreted as an OMS query and is passed to the OMS for evaluation. The resulting collection of PDM objects is destined for the Browser.
- PDM object

  The PDM object becomes destined for the Browser.
- QRL list

  A QRL list can contain any number of the above, and each element of the list is processed as defined above.

The browse function accumulates all of the PDM objects that are passed or collected as the result of a query, preserves the order in which they are passed, and, when the script finishes, constructs a fill-table and passes it to the tabular Browser for browsing by way of a **stpem** message. For more information about the Repository Browser, see *Fundamentals of StP*.

The display format for the PDM object's data is determined by how the browse function builds the fill-table, which is closely tied to the table types that are defined for the tabular Browser (tbr.rules).

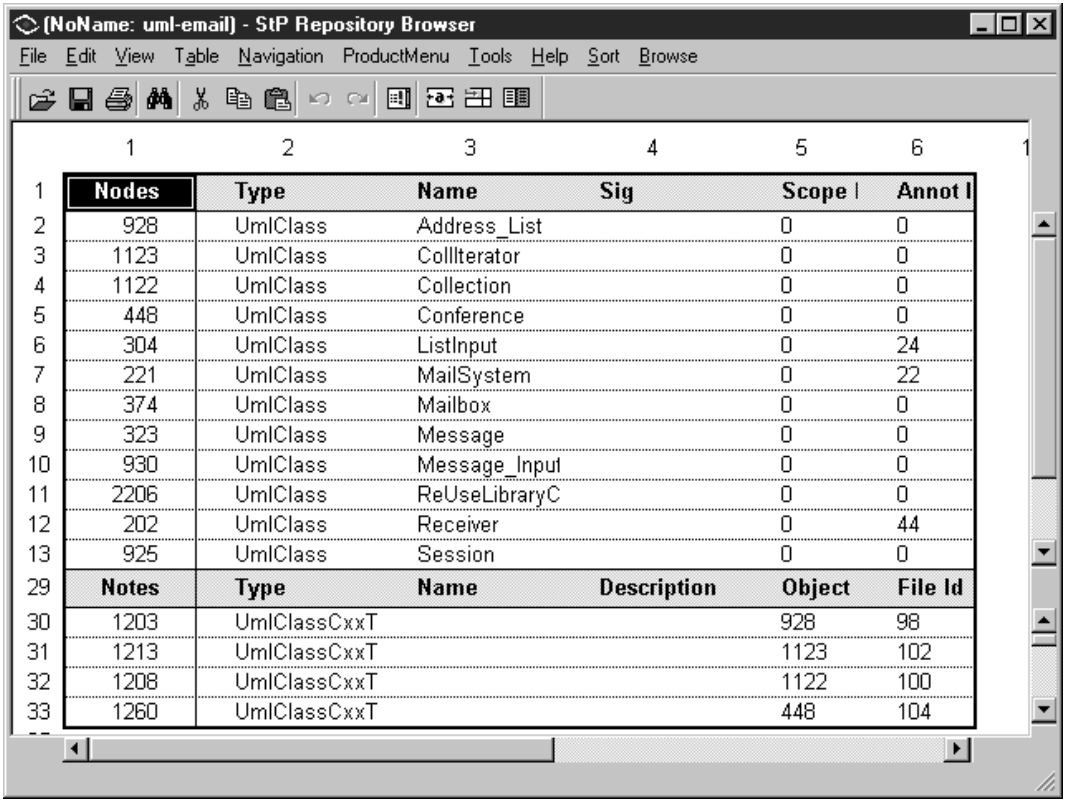
This example provides an alternative version of the example in Figure 3 on page 4-16. In that example, the print_line function was used to display the results of the script. This example substitutes the browse function for the print_line function, so that the output is displayed to the Repository Browser.

```
struct node_info
{
    node object;
    list note_list;
};
void
main()
{
    node_info node_struct;
    note note_var;
    node_struct.note_list = list_create("note", 0);
    for_each_in_select("node[UmlClass]" +
        "sort by name", node_struct.object)
    {
        browse(node_struct.object);
        for_each_in_select("note[obj_id == " +
```

```
            "${node_struct.object.id}]", note_var)
        {
            list_append(node_struct.note_list, note_var);
            browse(note_var);
        }
    }
}
```

The script invokes the Repository Browser, which displays the objects retrieved from the repository. A sample output of the script is given in Figure 4.

Figure 4: Repository Browser



	1	2	3	4	5	6
1	Nodes	Type	Name	Sig	Scope I	Annot I
2	928	UmlClass	Address_List		0	0
3	1123	UmlClass	CollIterator		0	0
4	1122	UmlClass	Collection		0	0
5	448	UmlClass	Conference		0	0
6	304	UmlClass	ListInput		0	24
7	221	UmlClass	MailSystem		0	22
8	374	UmlClass	Mailbox		0	0
9	323	UmlClass	Message		0	0
10	930	UmlClass	Message_Input		0	0
11	2206	UmlClass	ReUseLibraryC		0	0
12	202	UmlClass	Receiver		0	44
13	925	UmlClass	Session		0	0
29	Notes	Type	Name	Description	Object	File Id
30	1203	UmlClassCxxT			928	98
31	1213	UmlClassCxxT			1123	102
32	1208	UmlClassCxxT			1122	100
33	1260	UmlClassCxxT			448	104

Improving Script Efficiency

When you use OMS queries in a QRL script, focusing the scope of the query and taking other measures to avoid unnecessary trips to the repository speeds up processing time. This section contains suggestions for improving script efficiency.

Caching Objects

One way to eliminate multiple trips to the repository is to save objects in lists or sets, to iterate through the list or set as many times as needed, and then to free the list or set.

For example:

```
...
for_each_in_select("node[Process]", nd)
{
   jimbob(nd);
}
benny();
for_each_in_select("node[Process]", nd)
{
   freddy(nd);
}
...
```

Could be better written as:

```
...
nd_set = set_select("node[Process]");
for(i = 0; i < set_count(nd_set); i = i + 1)
{
   jimbob(set_get_element(nd_set, i));
}
benny();
loop_count = set_count(nd_set);
for(i = 0; i < loop_count; i = i + 1)
{
   freddy(set_get_element(nd_set, i));
}
```

```
set_clear(nd_set);
...
```

You can also use a list to cache objects if duplicate objects either do not exist or their existence does not constitute a problem. Using a list is more efficient than using a set.

In this case the selection is only performed once and remains memory resident until explicitly freed. For this reason you should be sure to free the set at the earliest possible time.

Using ID Lists

ID Lists are a way of constraining queries to a smaller set of objects than the entire repository. Consider using them in circumstances such as a template that performs repeated queries with common constraints. First, construct an ID list for all objects in the given diagram. Then use that ID list in all subsequent queries. Finally, free it immediately after its last use. See "Using the id_list_create and id_list_free Functions" on page 4-9 for an example of a script that uses ID lists.

Summary of Improving Script Efficiency

In general, beware of the following situations. If you find your code falls into one of these categories then carefully consider whether your approach is the best one.

- Nested queries.
- Queries that repeat elsewhere in the template. Consider saving the collection in a list or set.

5 Creating Documents

This chapter describes how to use QRL scripts to create, format, and print documents containing text and graphics. “Document” is used to refer to any output file that is the result of running a script (a document could be compilable C source code, for example). “Graphics” refers to tables and diagrams stored in the repository. The sections in this chapter are:

Using a QRL script you can:

- Extract information from the repository and include it in a document
- Create documents in ASCII, FrameMaker, HTML, or RTF
- Create your own document layout using format files

For information about writing a QRL script, see Chapter 3, “The StP Query and Reporting Language.” For information about using OMS queries in QRL scripts to extract information from the repository, see Chapter 4, “The OMS Query Language and QRL.”

In addition to being familiar with QRL, you should be familiar with a publishing tool: FrameMaker, Microsoft Word (or any other RTF-compatible publishing tool), or an HTML-compatible publishing tool.

Overview of Creating a Document

The simplest way to print a single diagram or table is with the **Print** command from the **File** menu of an StP diagram or table editor. The **Print** command is discussed in detail in *Fundamentals of StP*.

However, creating documents that include more complex elements, such as multiple diagrams and tables, included text files, and document formatting instructions, requires using a QRL script. This chapter describes how to do this.

For information on formatting and printing text in a document see "Printing Text" on page 5-9. For information on formatting and printing tables and diagrams, see "Printing Tables and Diagrams" on page 5-43.

"Extended Example" on page 5-79 shows how many of the procedures discussed in this chapter are combined in a complex script.

Using qrp

When you produce a document by running a script from the Script Manager, or with the **qrp** command, specifying the input file, publishing target, format file, and output file, the document is assembled dynamically from:

- Information in the script itself
- The repository, where the object information is stored
- A format file, which contains document-formatting information for the specified publishing target
- External variable values

You can either run your script from the graphical interface of the Script Manager, or with the **qrp** command. Chapter 3, "The StP Query and Reporting Language" provides a complete description of the **qrp** command, but a brief summary is given here for convenience.

Note: This chapter uses the **qrp** command for examples. For information on using the Script Manager instead of the **qrp** command, see Chapter 2, "The Script Manager."

The syntax for the **qrp** command is:

```
qrp <script_name> [-p <projdir>] [- s <system>]
    [-t <target>] [-f <format_file>]
    [-o <output_file>] [-I <include_dir>]...
    [-x <ext_var_name> <ext_var_value>]...
    [-trace <arguments>] [-trace_file <file>]
    [-version]
```

A complete list of **qrp** command options is given in Table 1. The options applicable for producing documents are shown in Table 1, below, for convenience.

Table 1: qrp Command Options for Formatting Documents

Option	Argument	Description
-f	<format_file>	Specifies the format file to be applied to the output. It can be a complete pathname, a relative path, or just a filename. If you specify only a filename or a relative path, the file is searched for in directories designated by the ToolInfo variable *<product>_stp_file_path*, with *print_format/<target>* added to the pathname.
-o	<output_file>	Specifies the full path and file to which you want to write the script's output (standard output is the default).
-t	**ascii** | **rtf** | | **html** | **mif** | **leaf**	Specifies the target output system. Can be RTF, HTML, FrameMaker, or ASCII. You do not need to specify **ascii**, since it is the default.

For example, to run a script, *script.qrl*, to create an ASCII document, *output.asc*, using only the system default formats, type:

```
qrp script.qrl -o output.asc
```

To format a FrameMaker document, *output.mif*, based on a FrameMaker format file, *report.mif*, type:

```
qrp script.qrl -t mif -f report.mif -o output.mif
```

To run the script to produce an RTF equivalent, type:

```
qrp script.qrl -t rtf -f report.rtf -o output.rtf
```

To run the script to produce an HTML equivalent, type:

```
qrp script.qrl -t html -f report.css -o output.html
```

Related Built-in Functions for Document Generation

The arguments for publishing target and format file have related built-in functions, shown in Table 2.

For a list of built-in functions specific to HTML, refer to "Additional Built-in Functions for HTML Document Generation" on page 5-5.

Table 2: Functions for Formatting Documents

Function	Return Type	Description
format()	string	Returns the name of the format file used to format the script's output.
format(string <format_file>)	string	Specifies the format file to be used to format the script's output. Returns the name of the format file. <format_file> may be a string or a string variable.
target()	target_ enum	Returns target specified for script output.

Regarding the format function:

- The format function that does not take an argument can be used anywhere in the script.
- The format function with a <format_file> argument must be called before you call any function that would make use of it.

  You can only use this type of format function call once in a script.

The command line option **-f** <format_file> overrides any specification in the script.

The argument of the format function can be either a string literal or a string variable. An example that uses a string literal to specify a format file is:

```
format("\u\me\my_formats.asc");
```

An example that uses a string variable is:

```
format(DefaultFormatFile);
```

In this case, the DefaultFormatFile variable must be defined in the script itself, or in an include file that is specified in the script. An example definition is:

```
const string DefaultFormatFile = "\u\me\my_formats.asc";
```

StP-supplied QRL scripts use include files with formats defined as string variables to make it easier to use your own document templates as format files. For information, see "StP-Supplied Format Include Files" on page 5-20.

Additional Built-in Functions for HTML Document Generation

HTML document generation provides additional built-in functions, allowing you to:

- Create document file names and titles.
- Generate Table of Contents, List of Diagrams, and List of Tables.

  Each entry in a generated list includes a hyperlink to the corresponding location in the document.

These functions are described in Table 3.

Table 3: HTML Functions for Formatting Documents

Function	Return Type	Description
document(string [file_name],string[title])	void	When called before any QRL output function is called, specifies the document's first output file name and title. When called after any QRL output function is called, begins a new document, using supplied arguments. If the file_name does not include a relative or absolute path, the file is placed in the same folder as the previous output file. See additional information following the table.
list_of_diagrams_print (string render_format)	void	Prints the List of Diagrams. Each entry corresponds to a diagram's caption. Text is rendered in the paragraph format supplied as the argument.
list_of_tables_print(string render_format)	void	Prints the List of Tables. Each entry corresponds to a table's caption. Text is rendered in the paragraph format supplied as the argument.
table_of_contents_entry (string entry_format, string render_format)	void	Specifies the paragraph format to be included and how its text is rendered in the Table of Contents. See additional information following the table.
table_of_contents_print()	void	Prints the Table of Contents. See additional information following the table.

Regarding the document function:

- If no output file is given for the first file, output goes to **stdout**.
- Providing an output file name on the command line overrides the <file_name> argument to document (for the first file only).
- If no initial <title> is provided, the name of the script is used.
- For all documents after the first file, if no arguments are supplied, **qrp** automatically generates the next file name and / or title.

  If a file name is generated automatically, an integer is added to the end of the file name (and before the extension), beginning with two (2). For example, if the initial output file name is *myDocument.html*, the next two automatically generated file names are:

  myDocument2.html
  myDocument3.html

  If a title is generated automatically, an integer is added to the end of the previous title, beginning with two (2). For example, if the title of the first document is "My Document Title," the automatically generated title for the next document is "My Document Title 2."

Regarding the table_of_contents_entry function:

- The function must be called before any QRL output function
- The following are examples using the table_of_contents_entry function:

```
table_of_contents_entry("H1", "UL");
table_of_contents_entry("H2", "LI");
table_of_contents_entry("ADDRESS", "LI");
table_of_contents_entry("P.class1", "LI");
```

Regarding the table_of_contents_print function:

- The function can be called anywhere within a script.
- If the function is called, each entry in the Table of Contents is generated when a call to the paragraph or page function is made, and the paragraph format was given as an *entry_format* argument to the table_of_contents_entry function (such as "H1" and "H2" in the above example).

  For information on the paragraph and page functions, see "Built-in Text Formatting and Printing Functions" on page 5-25.

How qrp Locates Format Files

If your script uses only the StP ASCII default formats, you do not need to specify a format file. If your script uses anything else (non-default ASCII formats, or any RTF, HTML, or FrameMaker formats), you must tell **qrp** which format file to use.

StP supplies the following format files with all StP products:

- *report* (FrameMaker)
- *basic* (RTF, HTML, and ASCII)
- *all_files*
- *pspec* (ASCII)

The *report* and *basic* format files are examples that contain a set of paragraph, character, and document formatting properties to use when creating your own report scripts.

For a list of StP product-specific format files, refer to the generating report section of your product documentation.

The format files supplied with StP are kept in the directories shown in Table 4. These directories are in the templates area of the directory specified by the ToolInfo variable <*product*>*_stp_file_path*.

Table 4: Location of Format Files Provided with StP

Target	Location of Format Files
RTF	*/templates/ct/print_format/rtf*
HTML	*/templates/ct/print_format/html*
FrameMaker	*/templates/ct/print_format/framemaker*
ASCII	*/templates/ct/print_format/ascii*

When looking for a format file, **qrp** looks in the directories in Table 4.

qrp adds the relative offset, *print_format/*<*target*>, to the value of *stp_file_path*. This means, for example, if you add this directory to *stp_file_path*:

/u/my_dir/project

qrp looks for the format file in:

/u/my_dir/project/print_format/<target>

Likewise, if the current directory is in the *stp_file_path* when you run **qrp**, the command looks for the format file in:

<current_directory>/print_format/<target>

To work around this, specify the file with a complete pathname, as:

```
./<filename>
```

To summarize:

- If your format file, *my_format*, is in a user-defined directory */u/me*, use:

  `-f ./my_format`—when you call **qrp** from */u/me*

  `-f /u/me/my_format`—when you call **qrp** from anywhere
- If *my_format* is in the *<product>_stp_file_path/templates/ct/print_format/<target>* directory, then no matter where you call **qrp** from, use:

  `-f my_format`
- If *my_format* exists in a user-defined directory, */u/me/print_format/<target>*, add */u/me* to *<product>_stp_file_path* and use (anywhere):

  `-f my_format`

To view the paths defined by the *<product>_stp_file_path* ToolInfo variable, type at the command line:

```
toolloc <product>_stp_file_path
```

For information about editing ToolInfo variable values, see *StP Administration*.

Printing Text

This section describes how to format and print text in a document that you produce with a QRL script. The topics covered in this section are:

- The StP defaults for formatting text
- The structure and syntax of RTF, HTML, FrameMaker, and ASCII format files
- The built-in functions for formatting text
- Example QRL scripts that format text to produce RTF, HTML, FrameMaker, and ASCII documents

What Format Files Do

A format file contains all of the information necessary for laying out a document. Format files enable you to specify attributes for:

- Entire documents
- Different types of paragraphs

  The term "paragraph" in this context refers to one or more characters ending in a carriage return, including bullets, titles, headings, and so on, but not words or phrases within a paragraph.
- A character or group of characters (word, phrase) within a paragraph that has different attributes (such as a different font) from the rest of the paragraph

  Character formats apply to non-ASCII formats only. In an ASCII file, for example, you cannot change the font of an individual character.

Different terms are used by the various publishing tools to refer to types of paragraphs and character definitions. FrameMaker uses the term "format," and Microsoft Word and HTML use "style." In this chapter, both "format" and "style" are used interchangeably.

Each of the four QRS publishing targets—RTF, HTML, FrameMaker, and ASCII—employs a different type of format file. ASCII uses an StP rules file; FrameMaker and RTF use documents created with the publishing tool; HTML uses Cascading Style Sheets (CSS).

Using Format Files

You use the paragraph and character formats that you define in the format file as arguments to functions in a QRL script. In addition, the format of the document as a whole is determined by the document format

information in the format file. When you run the script, the format definitions are imported from the format file into the output document.

Table 5 shows the way that format files and the document, paragraph, and character formats they contain map to each of the targets: RTF, HTML, FrameMaker, and ASCII.

Table 5: Mapping of Formats to Publishing Targets

Target	Format File	Document Format	Paragraph Format	Character Format
RTF	RTF document	Document format information in format file	Paragraph style from format file's style catalog	Character style from format file's style catalog
HTML	Cascading Style Sheet (CSS)		Paragraph style from CSS	Character style from CSS
FrameMaker	MIF document template		Paragraph format from format file's paragraph catalog	Character format from format file's character catalog
ASCII	StP rules file	Document rules	Paragraph rules	

QRS supplies format files for each of the publishing targets. These are distributed in the directories listed in Table 4 on page 5-8.

You can use the format files provided with StP, you can customize the StP-supplied ones, or you can create your own. The next section describes ASCII format files, followed by a section describing how to use them. Corresponding sections on RTF, HTML, and FrameMaker format files start with "Creating and Using Publishing Tool Format Files" on page 5-18.

Creating and Using ASCII Format Files

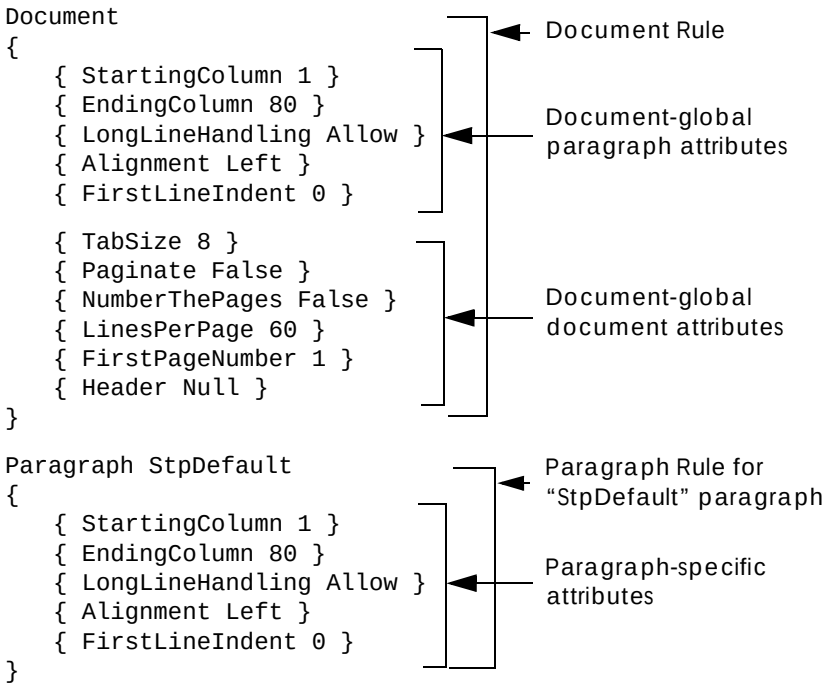
The default QRL publishing target is ASCII. The default definitions of ASCII documents and paragraphs are hard-coded into StP, but you can apply non-default definitions by using a format file that defines them.

The Structure of the ASCII Format File

The structure of an ASCII format file is an StP rules file. The file *example.asc* supplied with the system (see Table 4 on page 5-8 for its location) lists all the system defaults for ASCII formatting and illustrates the structure of an ASCII format file. This file is illustrated in Figure 1.

Note: QRL does not actually read this file to obtain its default formatting definitions. This file is provided only to show what the default settings are. To use values different from these settings, you create an ASCII format file that specifies the non-default values and use this file (either with the format function or the **-f** option on the command line) to make the changes take effect.

Figure 1: The *example.asc* File



Explanation:

- There are two types of rules in an ASCII format file: Document and Paragraph.

  A format file has one Document rule, but it can have as many Paragraph rules as needed.
- Document and Paragraph rules have attributes, such as StartingColumn and TabSize. Those attributes in turn have values; for example, TabSize has the value 8.
- Document rules define attributes that apply to the entire document; Paragraph rules define attributes of individual paragraph types (paragraph formats).
- Some attributes are only meaningful on a document level (Paginate, NumberThePages), and will only be used in a Document rule. Others can be used with both types of rules.
- Paragraph-type attributes set in the Document rule apply to all paragraphs in the document, unless you create a Paragraph rule with different values, and then apply it.
- Paragraph rules have names that are used as arguments to the paragraph function in a script. For example, the name of the Paragraph format in *example.asc* is "StpDefault." Document rules do not have names.
- When you use your own format file, you are in effect overriding default values. Whenever you do not specify a value for a particular attribute, the default value is used.

Table 6 lists all Document-only attributes and the values that can be legally assigned to them.

Table 6: Document-Only Attribute Values

Attribute	Permitted Values	Description
Paginate	True, False	Use True to start a new page after the number of lines specified by LinesPerPage, False for no page break.

Table 6: Document-Only Attribute Values (Continued)

Attribute	Permitted Values	Description
NumberThePages	True, False	Use True to add sequential page numbers to the bottom of each page, False for no page numbers. Only takes effect if Paginate is True.
LinesPerPage	A positive integer	Sets the number of lines on a page. Only takes effect if Paginate is True.
FirstPageNumber	An integer	Sets the page number of the first page of the document. Only takes effect if NumberThePages is in effect.
Header	Null, a string	Specifies text for a page header; Null means no header. Only in effect if Paginate is True.
TabSize	A positive integer	Specifies the space between tab stops (as on a typewriter) for all tabs, used to calculate the length of a line to determine where to wrap or truncate. Does not expand the tabs with spaces. Applies to left-aligned paragraphs only. A TabSize value for right- or center-aligned paragraphs is ignored.

Table 7 lists all Paragraph attributes and the values that can be legally assigned to them.

Table 7: Paragraph Attribute Values

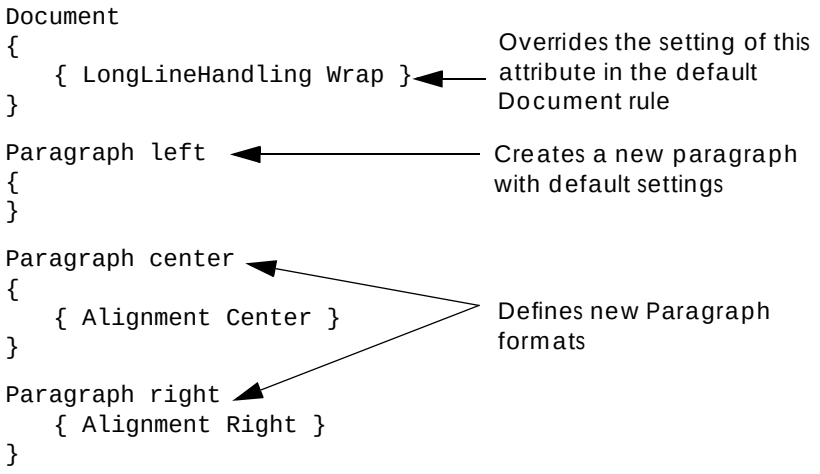
Attribute	Permitted Values	Description
StartingColumn	A positive integer	Sets the column for the left margin. Extreme left is 1.

Table 7: Paragraph Attribute Values (Continued)

Attribute	Permitted Values	Description
EndingColumn	A positive integer	Sets the column for the right margin.
LongLineHandling	Allow, Wrap, Truncate	Use Allow to allow a long line to extend beyond the set number of columns for a paragraph; use Wrap to wrap a long line to the next line; use Truncate to cut a long line after the last column is reached.
Alignment	Left, Center, Right	Sets the paragraph's alignment.
FirstLineIndent	An integer	Sets which column the first line of the paragraph is to start, relative to the left margin of a left-aligned paragraph or right margin of a right-aligned paragraph. Any value given for FirstLineIndent for a center-aligned paragraph is ignored.

Another ASCII format file supplied with StP is *basic.asc*. (See Table 4 on page 5-8 for location of this file.) Examining *basic.asc* illustrates how ASCII format files build on the internal defaults.

Figure 2: The *basic.asc* Format File



Explanation:

- This format file, *basic.asc*, does not repeat all of the information contained in *example.asc*, it only includes information that either adds to or overrides that information. When a script is run with *basic.asc*, any values not specified in *basic.asc* are taken from the internal defaults, as listed in *example.asc*.
- The *basic.asc* file defines three new paragraph formats: center, left, and right.

An example that shows how the paragraph formats in *basic.asc* are used in a QRL script is found in "Using Non-Default Formats" on page 5-30.

Creating an ASCII Format File

To create an ASCII format file:

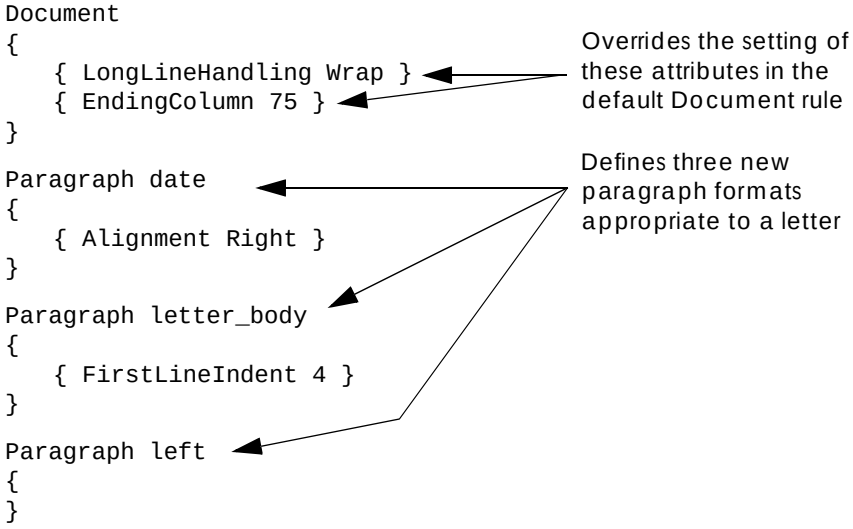
1. Open a text file using any text editor.
2. Enter the Document and Paragraph rules for which you want to add or change attributes and values from the internal defaults.

   For a review of the internal defaults, see Figure 1 on page 5-12.
3. Save the file.

It is recommended that you give the format file a *.asc* extension, such as *my_formats.asc*.

In the example in Figure 3, *my_formats.asc* file defines Document rules and Paragraph formats for a letter. It includes formats for a date (date), an indented paragraph (letter_body), and left alignment (left). It specifies an ending column size of 75 instead of the default 80, and Wrap as the value for LongLineHandling instead of the default Allow.

Figure 3: Example ASCII Format File, *my_formats.asc*



It is possible to create a format file that defines only Paragraph or only Document rules. Wherever values are not specified in the format file, default values are used.

An example that shows how the paragraph formats in *my_formats.asc* are used in a QRL script is found in Figure 10 on page 5-32. Information on how to specify which format file to use is given in "How qrp Locates Format Files" on page 5-8.

Creating and Using Publishing Tool Format Files

The basic principles for producing RTF, HTML, or FrameMaker documents with QRL are the same as for producing ASCII documents, with two exceptions:

- For RTF and FrameMaker, the format file is produced with the publishing tool: FrameMaker or Microsoft Word (or another publishing tool that produces RTF files), respectively.

  For HTML, the format file is produced with a text editor (or third-party HTML editor).
- Character formats can be defined in addition to document and paragraph formats.

RTF (Microsoft Word) Files

The format file used to produce a Microsoft Word file with QRL must be a document in RTF (Rich Text Format) format. RTF is a Microsoft-specified ASCII file format that allows exchange of information between Microsoft Word and other applications. RTF serves the same function for Microsoft Word that MIF does for FrameMaker.

You cannot use Microsoft Word templates as format files, since they cannot be saved as RTF files. Only actual documents can be saved as RTF files and used to provide formats to QRL scripts.

The document that you use to create the format file must contain examples of each of the character and paragraph styles required by the QRL scripts that will refer to them. Only the styles that actually appear in document content are guaranteed to be included in the style sheet that is created when a document is saved to RTF format.

For procedures for using Microsoft Word styles with QRL built-in functions, see "Creating Text Documents with QRL" on page 5-24.

For additional information on RTF documents, see "RTF Limitations" on page 5-69.

HTML Files

The format file used to produce an HTML file with QRL must be a Cascading Style Sheet (CSS) file. CSS files assign styles to HTML elements, including characters, paragraphs, and documents.

You can use any text editor or HTML-editor to produce CSS files. Each CSS file must end with a *.css* extension. If the specified CSS format file does not have a *.css* extension, one is automatically added.

The specified CSS file is incorporated into the document using the HTML LINK element. Document-wide format information is controlled by the CSS file. No specific document-wide formatting, such as attributes on the HTML BODY element, will be in the generated document itself.

For procedures for using HTML styles with QRL built-in functions, see "Creating Text Documents with QRL" on page 5-24.

FrameMaker MIF Files

The format file used to produce a FrameMaker file with QRL must be a FrameMaker document in MIF (Maker Interchange Format) format. MIF is a file format that allows exchange of information between FrameMaker and other applications.

A simple way to create a FrameMaker format file is to open a new FrameMaker document, create the paragraph and character formats your QRL script will need, and then save the document in MIF format. More details are described in "Creating Format Files With a Publishing Tool" on page 5-20.

When you create paragraph and character formats in FrameMaker, you identify each format by name. You use this name as the argument to a built-in QRL function to apply the format in a QRL script.

You do not have to create a special document for use as a FrameMaker format file; QRL can use format definitions from any FrameMaker file.

For procedures for using FrameMaker formats with QRL built-in functions, see "Creating Text Documents with QRL" on page 5-24.

The MIF version number of the output document will be the higher of 4 (the version used by QRS), and the version of the format file. To view a

MIF output document, the version of FrameMaker you use must be at least as high as the version of MIF in that document.

Creating Format Files With a Publishing Tool

Although you can view and edit an RTF file or FrameMaker MIF file with a text editor, you are more likely to add or modify document properties and paragraph and character formats / styles through the dialog boxes provided for that purpose by the publishing tool.

For HTML, it is recommended that you use a standard publishing tool, preferably one that includes dialog boxes for setting paragraph and character styles.

Information on how to specify which format file to use with a QRL script is given in "How qrp Locates Format Files" on page 5-8.

To create a format file:

1. Open a document in the publishing tool.
2. Use the publishing tool's menus and dialog boxes to add or change document attributes, paragraph formats, and character formats.
3. Save the file as an RTF file (Microsoft Word), CSS (HTML), or MIF file (FrameMaker).

   If you are creating an RTF, HTML, or FrameMaker format file, stop here.

StP-Supplied Format Include Files

StP uses format include files for managing the format names used by StP-supplied QRL scripts. This makes it easier to use your existing document templates as format files for QRL scripts.

StP supplies format include files that map QRL string variables to format names in a format file. Table 8 lists the locations of the format include files installed with all StP products and shows their corresponding QRL scripts and format files.

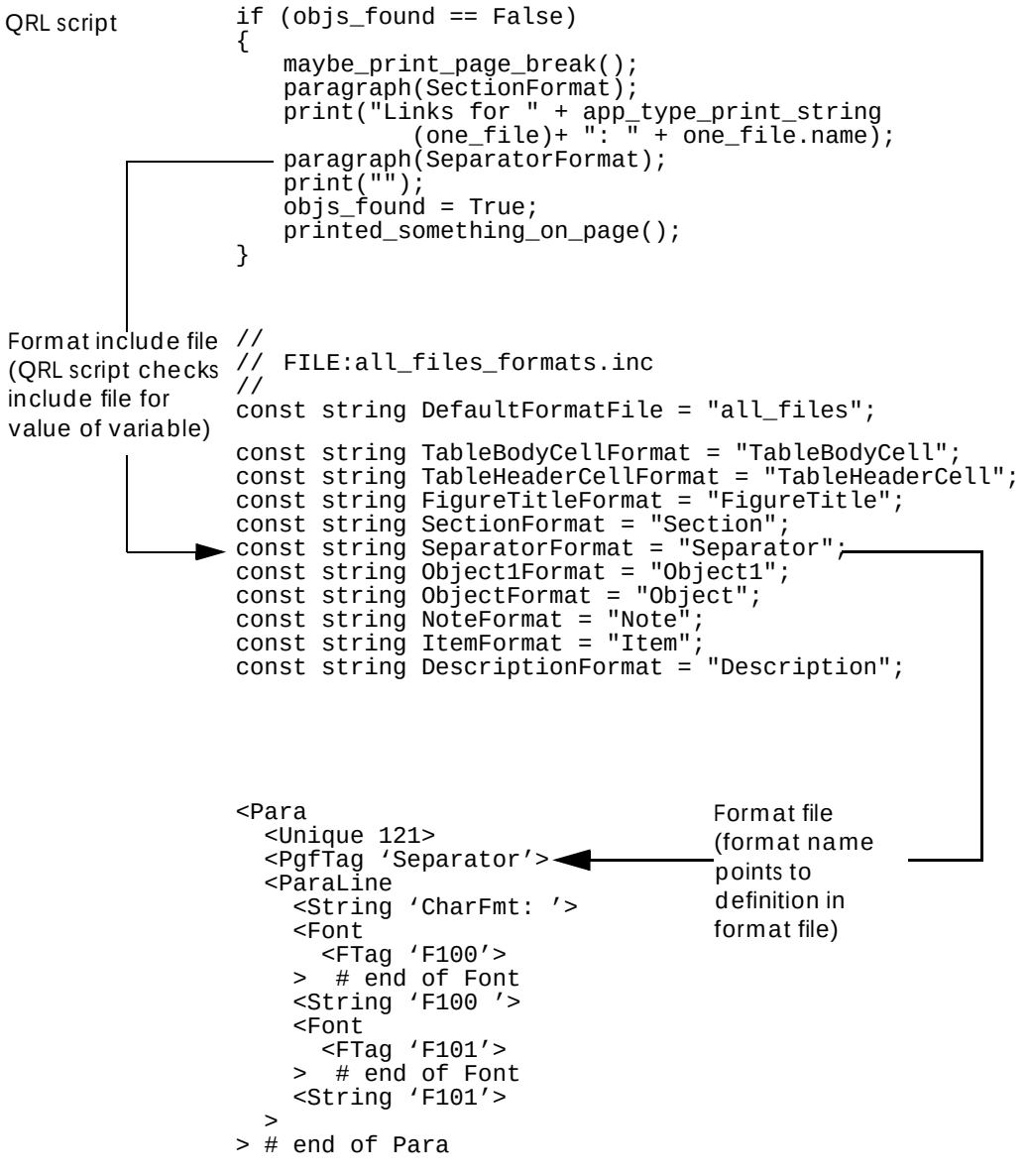
Table 8: Location of StP-Supplied Format Include Files

QRL Script	Include File	Format File
all_files.qrl	*templates/ct/qrl/include/all_files_formats.inc*	*all_files*
annotation.qrl	*templates/ct/qrl/include/annotation_formats.inc*	

For a list of StP product-specific format include files, refer to the generating report section of your product documentation.

Figure 4 on page 5-22 illustrates the relationship between the format include file (*all_files_formats.inc*) and its corresponding QRL script (*all_files.qrl*) and format file (in this case, a MIF format file).

Figure 4: QRL Script, Format Include File, and Format File



Customizing StP-Supplied Format Include Files

To use your own document template as a format file for an StP-supplied QRL script, modify the script's format include file, replacing the format names in the include file with the format names from your template.

To customize an StP-supplied format include file to use your own format file:

1. Using a text editor, open the StP-supplied format include file that you wish to edit.

   For the list of StP-supplied format include files, see "StP-Supplied Format Include Files" on page 5-20.

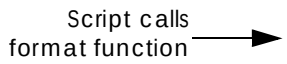
2. Edit the values of the character and paragraph formats, changing them to the names of formats in your template.

   Use the exact spelling and case as used in your format file.

3. If the corresponding QRL script makes a call to the format function, replace the value of the variable defining the format file with the name of your format file.

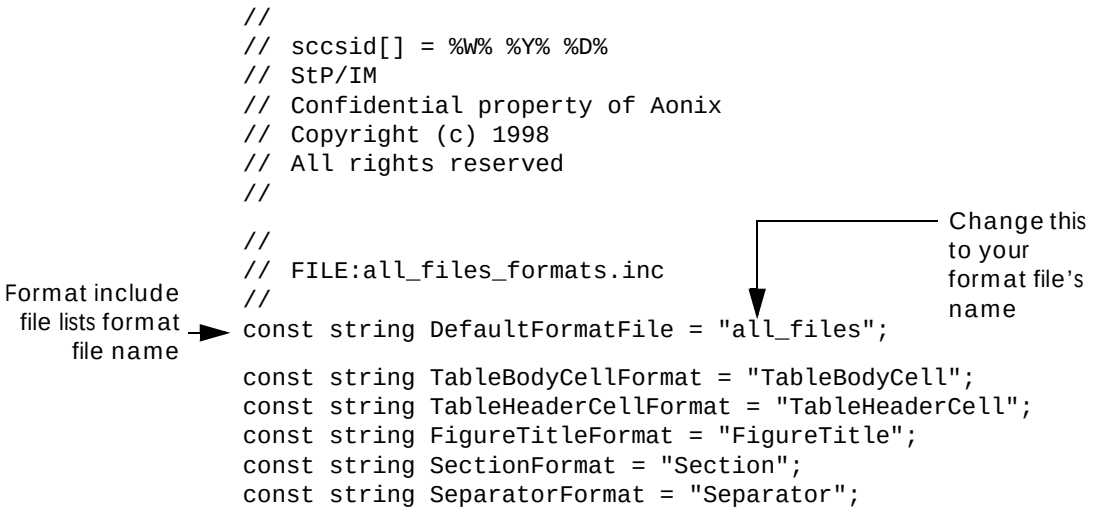
   The *all_files.qrl* script shown in Figure 5 makes a call to the format function. The corresponding format include file, *all_files_formats.inc*, is shown in Figure 6. In this example, you would change value "all_files" to the name of your format file.

Figure 5: Section of the *all_files* QRL script




```
// set default format
format(DefaultFormatFile);
table_cell_paragraph_format_set(TableBodyCellFormat);
table_cell_bold_paragraph_format_set
(TableHeaderCellFormat);
table_caption_paragraph_format_set(FigureTitleFormat);
diagram_caption_paragraph_format_set(FigureTitleFormat);
```

Figure 6: *all_files_formats.inc* Include File (Partial)




```
//
// sccsid[] = %W% %Y% %D%
// StP/IM
// Confidential property of Aonix
// Copyright (c) 1998
// All rights reserved
//

//
// FILE:all_files_formats.inc
//
const string DefaultFormatFile = "all_files";

const string TableBodyCellFormat = "TableBodyCell";
const string TableHeaderCellFormat = "TableHeaderCell";
const string FigureTitleFormat = "FigureTitle";
const string SectionFormat = "Section";
const string SeparatorFormat = "Separator";
```

There should be no more than one include file per script. However, scripts can share the same format include file. The name of the include file should be in the format: <*name*>*_formats.inc*.

Creating Text Documents with QRL

QRL includes functions that enable you to format and print text. For ASCII text, you can apply paragraph formats. For Microsoft Word, HTML, or FrameMaker text, you can apply both paragraph and character formats.

This section describes:

- The QRL built-in text formatting and printing functions
- ASCII, Microsoft Word, HTML, and FrameMaker examples

Built-in Text Formatting and Printing Functions

The text-formatting functions of QRL perform such tasks as:

- Printing text
- Inserting tabs and newlines
- Skipping lines
- Forcing page breaks
- Applying text formats

These functions are listed in Table 10. Explanations and examples follow.

Table 10: Text Formatting and Printing Functions

Function	Return Type	Description
character_format_set (string <character_format>)	string	Changes character format of text; returns current character format.
page()	string	Starts a new page, using the paragraph format specified for the previous paragraph. Returns the previous paragraph format.
page (string <paragraph_format>)	string	Starts a new page using the specified paragraph format. Returns the previous paragraph format.
page_equals_document (target_enum a_target)	void	HTML Only. Specifies that each call to page is identical to a call to document with empty arguments (see Table 3 on page 5-6).

Table 10: Text Formatting and Printing Functions (Continued)

Function	Return Type	Description
paragraph()	string	Starts a new paragraph using the paragraph format specified for the previous paragraph format. If none was specified, uses the default. Returns the previous paragraph format.
paragraph (string <paragraph_format>)	string	Changes paragraph format and starts a new paragraph. If the paragraph definition includes starting a new page, then this creates a page break. Returns old paragraph format.
paragraph_end()	void	Ends the current paragraph. Necessary for generating nested HTML lists. For an example, see "HTML Example" on page 5-36
print(<all value>)	void	Prints the value where <value> can be any legal expression. Does not add a hard return. This is the preferred function to use with paragraph formats that wrap (such as RTF, HTML, and FrameMaker paragraph formats, and ASCII paragraphs if LongLine-Handling is set to Wrap).
print_line()	void	Starts a new line.
print_line(<all value>)	void	Prints the value and starts a new line; where <value> can be any legal expression.

Table 10: Text Formatting and Printing Functions (Continued)

Function	Return Type	Description
print_link(<all types> <value>, string <mark_label>)	void	HTML only. Prints values in the identical manner as the print function. In addition, the value becomes the source of an HTML hyperlink whose destination is given by the string mark_label.
print_mark(<all types> <value>, string <mark_label>)	void	HTML only. Prints values in the identical manner as the print function. In addition, the value becomes a mark, that is, an anchor to serve as the destination of an HTML hyperlink. Each mark must have a unique mark_label.
print_raw(string <value>)	void	For non-ASCII output (that is, RTF, HTML, and FrameMaker), does not add any formatting statements to the output document, and prints the value without escaping any character sequences meaningful to the target. (**Note:** all other print commands escape meaningful sequences.)
printf(string format,... list of expressions)	void	Like C **printf** function. Translates values to characters. Converts, formats, and prints the values as substituted in arguments prefixed by %. QRL supports most of the same formatting characters and options as the C language.

Table 10: Text Formatting and Printing Functions (Continued)

Function	Return Type	Description
skip_line(int <lines>)	void	Skips number of <lines> specified in argument.
tab(int <tabs>)	void	"Tabs" over the number of tab stops specified in argument.

Creating Simple Documents Using System Defaults

The simplest script you can write to format text is one that uses only StP (ASCII) defaults. Two examples using ASCII defaults are offered in this section: One calls an OMS query and uses the print_line function to format the output, the other illustrates use of several text formatting functions.

This sample calls an OMS query to retrieve all StP/UML Class Editor diagram files from the repository and then prints their names in a list. The script uses only system (ASCII) defaults and the single built-in function print_line.

```
void
main()
{
    file file_var;
    // Print literal string
    print_line("This is a list of all of the UML " +
        "Class diagrams on the system:\n");

    // Retrieve all Class diagram files from repository
    for_each_in_select("file[UmlClassDiagram]",
        file_var)

        // Print name field of file_var
        print_line(file_var.name);
}
```

You can run this script by typing:

```
qrp script.qrl -o simple_report.txt
```

In the above example, *script.qrl* is the script and *simple_report.txt* is the output file. Notice that there is no target specification, because ASCII is the default.

Depending on the names of the Class diagrams on your system, the contents of *simple_report.txt* resemble the following:

```
This is a list of all of the UML Class diagrams on the
system:

<diagram_name_1>
<diagram_name_2>
.
.
.
```

The next sample script formats a simple ASCII document using several more functions from Table 10 on page 5-25.

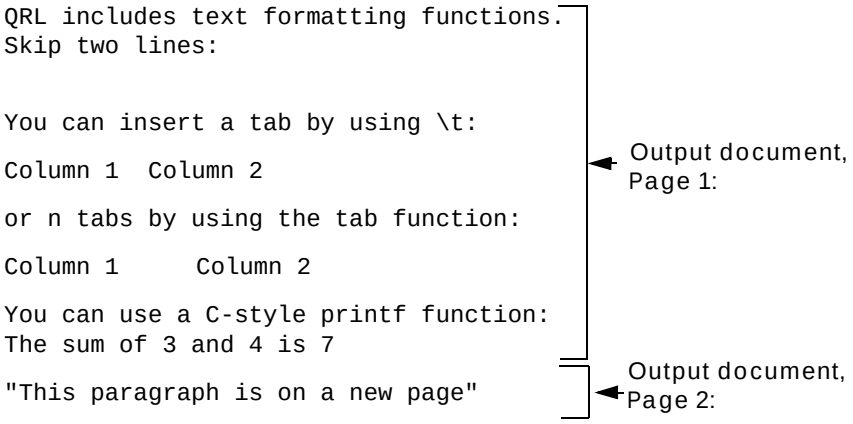
```
void
main()
{
    // Initialize variables to be used later in script
    int val1 = 3, val2 = 4, sum;
    sum = val1 + val2;

    print_line("QRL includes text formatting " +
       "functions.");
    print_line("Skip two lines:");
    skip_line(2);
    print_line("You can insert a tab " +
       "by using \\t:");
    skip_line(1);
    print_line("Column 1 \t Column 2");
    print_line("\n");
    print_line("or n tabs by using the tab function: \n");
    print("Column 1");
    tab(2);
    print_line("Column 2\n");

    // Use C-style printf function to print integer values
    // of val1, val2, and sum
    print_line("You can use a C-style printf function:");
    printf("The sum of %i and %i is %i\n",
       val1, val2, sum);
    // Force a page break
```

```
    page();
    print_line("This paragraph is on a new page.");
}
```

In this script, the page() function (third line from the end) should start a new page using a paragraph format specified in a previous function call. Since no argument is given and no other paragraph is specified in the script, the default paragraph format is used. (This is the default ASCII format defined internally and stated explicitly in the *example.asc* file in Figure 1 on page 5-12.)

When run, this script produces the document shown in Figure 7.

Figure 7: Output of ASCII Text Formatting Script



Using Non-Default Formats

The example script in Figure 8 refers to a non-default ASCII format file, *basic.asc*, shown in Figure 2 on page 5-16. This script shows how non-default ASCII paragraph formats can be used in a script to format its output.

When you cite the name of a paragraph as an argument, the case and spelling of the argument must exactly match the corresponding paragraph format name listed in the format file (for example, `paragraph("center")`).

When you use paragraph() without an argument before a print statement, a new paragraph is started that uses the format from the previous paragraph.

Figure 8: Sample Script: Using Non-Default ASCII Formats

```
void
main()
{
    // Center paragraph
    paragraph("center");
    print_line("This paragraph is centered.");

    // Make paragraph left-justified
    paragraph("left");
    print_line("Skip two lines:");
    // Use skip_line function to skip number of lines
    // specified in argument (in this case, 2)
    skip_line(2);

    // Make paragraph right-justified
    paragraph("right");
    print_line("This paragraph is right-justified.");

    // Start a new paragraph; use format of previous
    // paragraph
    paragraph();
    print_line("And so is this one.");

    // Return text to left
    paragraph("left");
    print_line("Now everything is back on the left.");
}
```

Using paragraph format "center"

Using paragraph format "left"

Using paragraph function without an argument

Run this script by typing:

```
qrp script.qrl -f basic.asc -o example.txt
```

Figure 9 shows the document produced by this script.

Figure 9: Output of Script Using Non-Default ASCII Formats

```
                This paragraph is centered.
Skip two lines:

                          This paragraph is right-justified.
                                       And so is this one.

Now everything is back on the left.
```

Reading Text in From a File

All preceding examples have provided text as a string argument to functions within the script. The example shown in Figure 10 uses the read_file function to provide text from a file, in this case, the *letter_body.txt* file.

The paragraph formats used in this sample (date, letter_body, and left) are defined in *my_formats.asc* from Figure 3 on page 5-17.

Figure 10: Sample Script: Reading Text in From a File

```
void
main()
{
   string inc_file;

   // Apply "date" paragraph format; causes text to be
   // right-justified
   paragraph("date");
   print_line("December 22, 1998");
   // Apply "left" paragraph format; causes text to be
   // left-justified
   paragraph("left");
   print_line("Dear Ms. Smith,");

   // Use read_file to include a file in the document
   inc_file = read_file("letter_body.txt");

   // Apply "letter_body" paragraph format to included
   // file, inc_file
   paragraph("letter_body");
   print(inc_file);
   // Apply "left" paragraph format
   paragraph("left");
```

```
    print_line("Sincerely,");
    skip_line(1);
    print_line("Mr. Jones");
}
```

Assume the format file, *my_formats.asc*, exists in the */u/jones* directory. This is not a standard format file directory (refer to "How qrp Locates Format Files" on page 5-8). To run this script, specify the location of the format file, by typing:

```
qrp script.qrl -f /u/jones/my_formats.asc -o letter
```

The output document *letter* is shown in Figure 11.

Figure 11: Output of Script Reading in Text From a File

```
                                                October 22, 1998

Dear Ms. Smith,

   I am writing to inform you that your car is being
repossessed. The insurance companies have not been able to
reach a settlement. Please let me know what would be a good
time to pick up the car.

Sincerely,

Mr. Jones
```

Formatting HTML, FrameMaker, or Microsoft Word Documents

To create scripts to produce HTML, FrameMaker, or Word documents:

- Use the print function rather than print_line.

  print_line creates a hard Return (starts a new line). Since FrameMaker, HTML, and Word paragraph formats automatically start new lines, this is unnecessary.

  You can achieve the same behavior in ASCII output by using a LongLineHandling value of Wrap.
- Use paragraph() without an argument before a print statement to start a new paragraph using the format from the previous paragraph.

- For all publishing tools except Microsoft Word, use the exact case and spelling in the argument supplied to the paragraph function as the corresponding paragraph format in the format file used by the script.

  For example, `paragraph("Title")`
- For all publishing tools except Microsoft Word, use the exact case and spelling in the argument supplied to the character_format_set function as the corresponding character format in the format file used by the script.

  For example, `character_format_set(“F12”)`
- For Microsoft Word only, to avoid unexpected results, be aware that some Word paragraph styles are placed in a fixed position on the current page rather than following sequentially after the previous paragraph.

  For example, in the Microsoft Word template “Professional Report,” the Company Name and Part Title styles occupy a fixed position on the page that cannot be modified.

To run scripts to produce HTML, FrameMaker, or Word documents, use the same rules as with formatting ASCII documents except:

- You must specify a format file; there are no HTML, FrameMaker, or Word defaults
- You must specify a target

Microsoft Word Example

The sample script in Figure 12 uses paragraph and character formats from the *basic* file. Paragraph formats that are defined in *basic* and used in this script include:

- Title
- Heading 1
- Body
- Bullet

The character format used in this script and defined in *basic* is “Emphasis,” which is italic.

Figure 12: Sample Script: Using an RTF Format File

```
void
main()
{
    // Apply paragraph format "Title"
    paragraph("Title");
    print("This is the title");

    // Apply paragraph format "Heading 1"
    paragraph("Heading 1");
    print("And This is a Level 1 Heading");

    // Apply paragraph format "Body"
    paragraph("Body");
    print("This is what body text looks like.");

    // Start a new paragraph; continue using format
    // applied to previous paragraph
    paragraph();
    print("You can use bullets:");

    // Apply paragraph format "Bullet"
    paragraph("Bullet");
    print("Text for first bullet.");
    paragraph();
    print("Text for second bullet.");

    // Apply paragraph format "Body"
    paragraph("Body");
    print("Or change a single ");

    // Use the character_format_set function to set a
    // character format for a particular string
    character_format_set("Emphasis");
    print("word.");
}
```

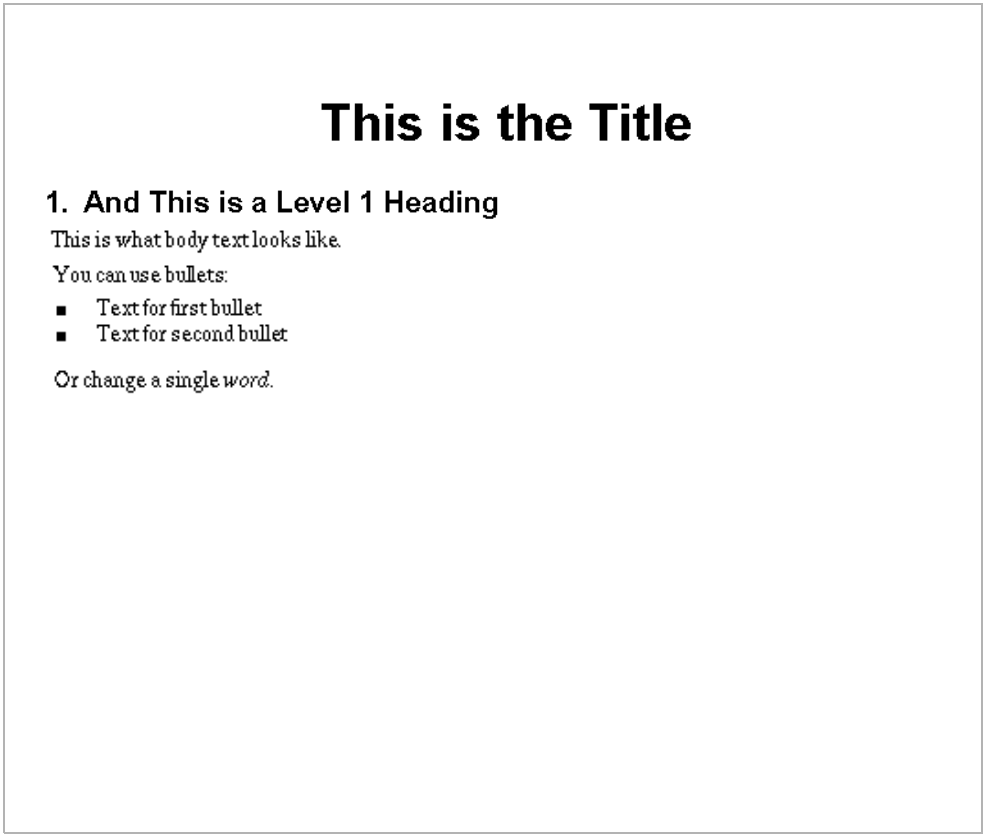
Run this script by typing:

```
qrp script.qrl -t rtf -f basic -o output.rtf
```

where *script.qrl* is the script and *output.rtf* is the output file.

The resulting document looks like Figure 13 when you open it in Microsoft Word.

Figure 13: Document Produced with *basic* Format File



For additional information on autonumber limitations with RTF, see "Autonumbered Paragraph Formats and RTF" on page 5-41.

HTML Example

The sample script in Figure 14 uses paragraph and character formats from the *basic.css* file. Paragraph formats that are defined in *basic.css* and used in this script include:

- ADDRESS
- ADDRESS.class1
- P

Additional HTML-defined paragraph tags used in this scripts include H1, UL, LI, OL, DL, and DT. The character formats used in this script and defined in *basic* are:

- EM.class1
- SPAN

Figure 14: Sample Script: Using an HTML Format File

```
void
main()
{

    paragraph("ADDRESS");
    print_line("This is ADDRESS text");

    character_format_set("EM.class1");
    print_line("This is the EM.class1 character tag");

    paragraph("H1");
    print("This is an H1 heading");

    paragraph("ADDRESS.class1");
    print_line("This is ADDRESS.class1 text");

    character_format_set("SPAN");
    print_line("This is the SPAN character tag");

    paragraph("P");
    print_line("This is the P paragraph tag");

    paragraph();
    print_line("You can use bullets:");

    // List definition and list item elements
    paragraph("UL");
    paragraph("LI");
    print("first outer list item");
    paragraph("LI");
    print("second outer list item");
    paragraph("OL");
    paragraph("LI");
    print("first inner list item");
    paragraph("LI");
```

```
    print("second inner list item");
    paragraph_end();
    paragraph("LI");
    print("third outer list item");
    paragraph("DL");
    paragraph("DT");
    print("text for DT");
    paragraph("DD");
    print("text for DD");

}
```

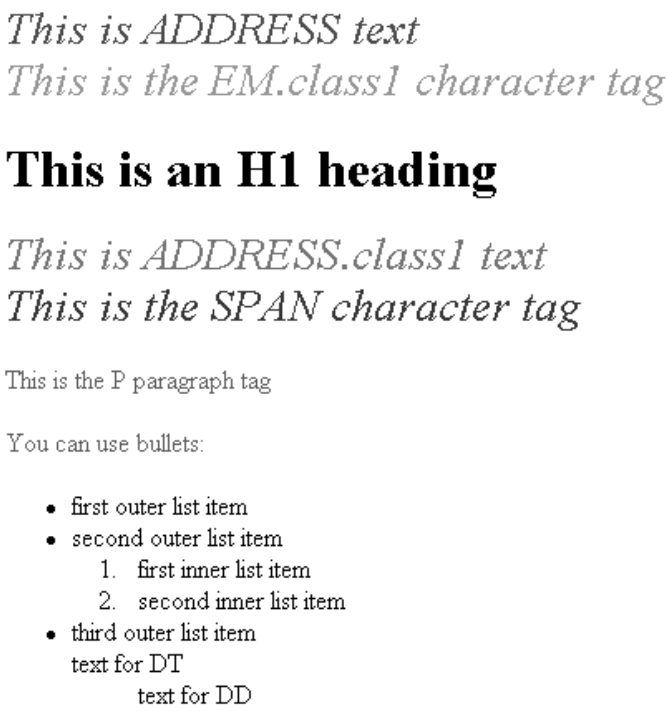
Run this script by typing:

```
qrp script.qrl -t html -f basic.css -o output.html
```

where *script.qrl* is the script and *output.html* is the output file.

The resulting document looks like Figure 15 when you open it in a web browser.

Figure 15: Document Produced with *basic.css* Format File



HTML document generation includes the ability to generate a Table of Contents, List of Diagrams, and List of Tables. For the description of these functions, see "Additional Built-in Functions for HTML Document Generation" on page 5-5.

FrameMaker Examples

The sample script in Figure 16 uses paragraph and character formats from the *report.mif* file. This file is a FrameMaker format file provided with StP. (The location of this file is *<product>_stp_file_path/templates/ct/print_format/framemaker.*)

Paragraph formats that are defined in *report.mif* and used in this script include:

- Title
- 1Heading
- Body
- Bullet

The character format used in this script and defined in *report.mif* is "Emphasis," which is italic.

Figure 16: Sample Script: Using a FrameMaker Format File

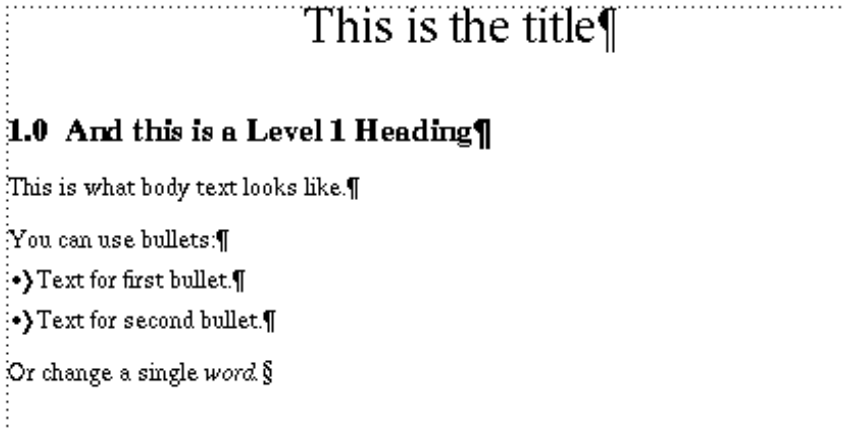
```
void
main()
{
    // Apply paragraph format "Title"
    paragraph("Title");
    print("This is the title");

    // Apply paragraph format "1Heading"
    paragraph("1Heading");
    print("And This is a Level 1 Heading");

    // Apply paragraph format "Body"
    paragraph("Body");
    print("This is what body text looks like.");

    // Start a new paragraph; continue using format
    // applied to previous paragraph
    paragraph();
    print("You can use bullets:");

    // Apply paragraph format "Bullet"
    paragraph("Bullet");
    print("Text for first bullet.");
    paragraph();
    print("Text for second bullet.");

    // Apply paragraph format "Body"
    paragraph("Body");
    print("Or change a single ");

    // Use the character_format_set function to set a
    // character format for a particular string
    character_format_set("Emphasis");
    print("word.");
}
```

Run this script by typing:

```
qrp script.qrl -t mif -f report.mif -o output.mif
```

The resulting document looks like Figure 17 when you open it in FrameMaker.

Figure 17: Document Produced with *report.mif* Format File



Autonumbered Paragraph Formats and RTF

RTF and Microsoft Word have more limited autonumbering capabilities than FrameMaker, and require more tools from QRL to achieve the same level of functionality.

This section describes how to generate autonumbered:

- Headings
- Lists
- Diagram captions
- Table captions

Autonumbering in Headings and Lists

You can create autonumbered headings and lists by using the appropriate paragraph format in the format file. For headings, Microsoft Word

(English version) requires that you use specific names for the heading format, namely, *Heading 1*, *Heading 2*, and so forth, up to *Heading 9*. For lists, however, you can use any name.

Autonumbering in Diagram and Table Captions

QRL uses Seq (Sequence) fields for autonumbering diagram and table captions the same way Microsoft Word does in its "Insert Caption" capability. In addition to Sequence fields, you can use a StyleRef field to include a heading number as a prefix to the Sequence number. Microsoft Word also has this capability. For information on Sequence and StyleRef fields, see your Microsoft Word documentation.

You control QRL's use of Sequence and StyleRef fields with several ToolInfo variables, listed and described in Table 11.

Because QRL uses Sequence fields for captions, do not include autonumbering in the diagram and table caption styles in your format file. This would be redundant and cause conflicts with QRL's mechanisms.

Table 11: ToolInfo Variables for RTF Autonumbering

Variable	Type	Default Value	Description
qrs_rtf_use_seq_field	Boolean	yes	Specifies whether or not Sequence fields are used.
qrs_rtf_diagram_seq_field	String	Figure	Specifies the label and identifier for diagram caption Sequence fields.
qrs_rtf_table_seq_field	String	Table	Specifies the label and identifier for table caption Sequence fields.
qrs_rtf_use_styleref_field	Boolean	no	Specifies whether or not a StyleRef field is used. The referenced style is always "Heading 1."

Autonumbered Paragraph Formats and HTML

The current version of Cascading Style Sheets, CSS Level 1.0, does not support autonumbered paragraph formats.

Printing Tables and Diagrams

This section describes how to produce a Microsoft Word, HTML, or FrameMaker document with a QRL script that includes tables and diagrams from the repository.

The topics covered in this section are:

- Understanding print options and print settings
- Specifying the basic unit of measure
- Including tables or diagrams in a document
- Modifying print option values
- Diagram and table formatting limitations
- Customizing shading and color in a table

Understanding Print Options and Print Settings

A print option specifies a single aspect of the appearance of a diagram or table. For example, the width and height of the frame a diagram is to be printed in are individual print options. A print setting is a collection of print options and specifies the complete appearance of a diagram or table.

There are built-in QRL functions that are used to set values for individual print options. These functions are described in "Modifying Print Option Values" on page 5-49.

To change the default values of print options, see *Customizing StP*.

Named Settings

When you print a diagram or table from an editor, the **Print** command's dialog box displays the print options available for printing the object. You

can modify the option values and save them as a "named setting." You can create multiple named settings for a single diagram or table. A named setting can only be used with the table or diagram it is associated with; it cannot be used with any other diagram or table.

Details for creating a named setting are described in *Fundamentals of StP*. For information on how you use named settings in a QRL script, see "Using Named Settings" on page 5-70.

Specifying the Basic Unit of Measure

A unit of measure is required for functions that specify sizes and positions. QRL includes the file_print_units_set function for specifying this basic unit. This is an overloaded function, shown in Table 13.

Table 12: Function for Specifying the Unit of Measure

Function	Return Type	System Default	Description
file_print_units_set (units_enum <units>)	units_ enum	Inches	Sets the units as either inches (Inches) or centimeters (Cm).
file_print_units_set (float <points_per_unit>)	float	72	Sets the number of points per unit when a unit other than Inches or Cm is desired.

The first form of this function sets the units either as inches or centimeters:

```
units_enum old_units;

old_units = file_print_units_set (Inches);
old_units = file_print_units_set (Cm);
```

The second form of this function sets the number of points in any unit. This example has the same effect as the previous example, since there are 72 points per inch and 28.346 points per centimeter:

```
float old_units;
```

```
old_units = file_print_units_set (72.0);
old_units = file_print_units_set (28.346);
```

Including Tables and Diagrams in Documents

Table 13 lists the built-in QRL functions for including tables and diagrams that exist in the repository in a QRL script.

Table 13: Functions for Printing Diagrams and Tables

Function	Return Type	Description
file_print (file <file_object>, string <caption>)	void	Prints a table or diagram where <file_object> must be a table or diagram file.
file_print (file <file_object>, string <caption>, string <named_setting>)	void	Prints a table or diagram where <file_object> must be a table or diagram file. Uses the specified <named_setting> instead of the script's option values.
file_print_named_settings_use (boolean <value>)	void	Turns on or off the ability of a script to use the named setting associated with the table or diagram being printed.

The file_print function is an overloaded function. Its simplest form is:

```
void file_print(file <file_object>, string <caption>)
```

where <file_object> is the diagram or table file, and <caption> is its caption text. If you do not want to include a caption for the table or diagram, use an empty string ("") as the <caption> argument. If you want the caption text to come from a named setting or default value, use NULL as the <caption> argument.

The second form of file_print is:

```
void file_print(file <file_object>, string <caption>,
   string <named_setting>)
```

Use this form when you want to apply a named setting to the object. The <named_setting> argument is a named setting name. This use of file_print and file_print_named_settings_use are discussed fully in "Using Named Settings" on page 5-70.

The example script in Figure 18 on page 5-47 uses the file_print function to print the UML diagrams returned by the for_each_in_select function. Note that the argument to the file_print function is a file object in the repository. Diagrams and tables must be stored in the repository before they can be included in QRS documents.

Including Tables and Diagrams in a MIF document

Figure 18 on page 5-47 provides an example script, *script.qrl*, that generates a MIF document containing a diagram.

For a description of functions used to set diagram print options (such as the ones in *script.qrl* that set the diagram orientation and caption format), see "Modifying Print Option Values" on page 5-49.

You run *script.qrl* by typing:

```
qrp script.qrl -t mif -f report.mif -o uml_diag.mif
```

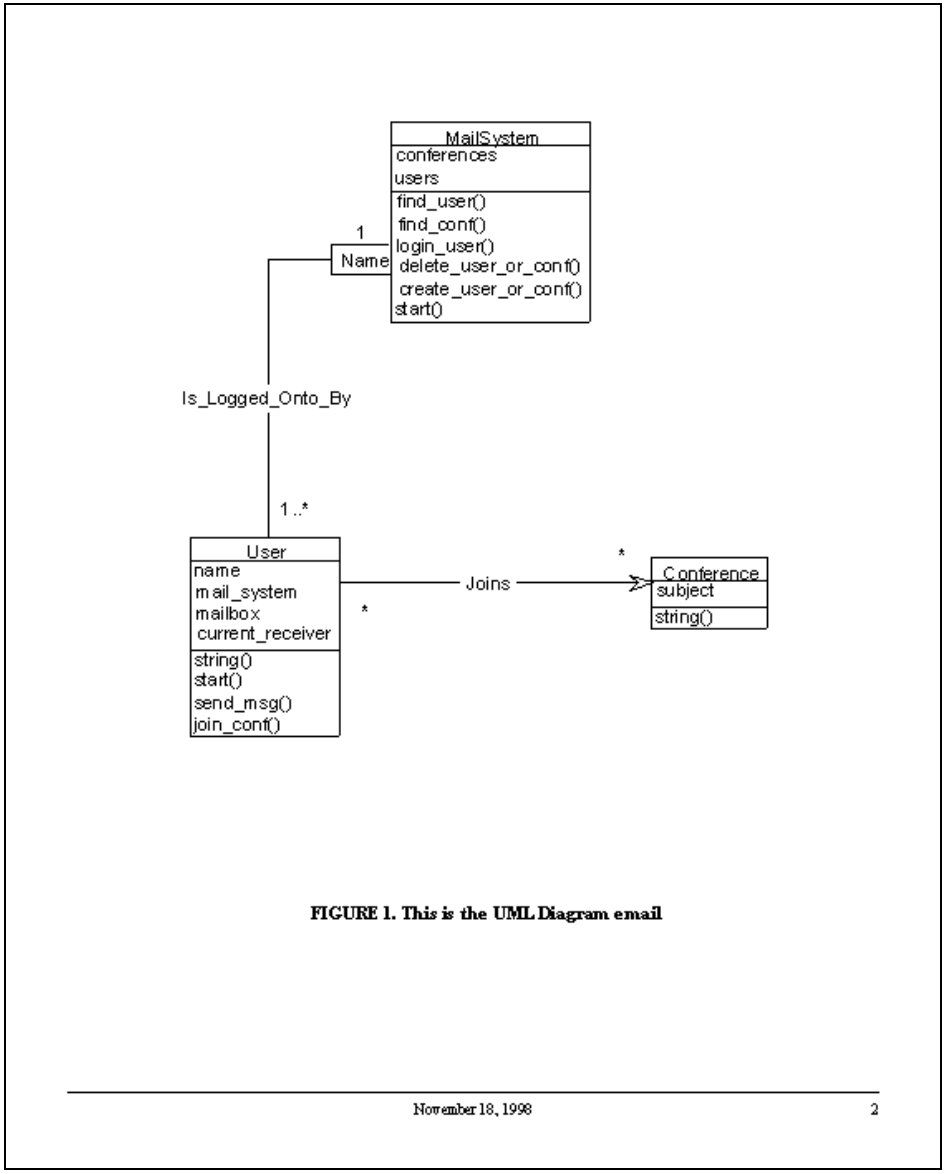
where *script.qrl* is the script name, mif is the target, *report.mif* is the format file, and *uml_diag.mif* is the output file.

Figure 18: Sample Script: file_print Function

```
void
main()
{
    file file_var;
    string caption;

    // Apply paragraph format "Title" to title
    paragraph("Title");
    print("UML Diagrams Report");

    // Force a page break after the title page
    page();

    // Apply paragraph format "Figure" to diagram
    // caption
    diagram_caption_paragraph_format_set("Figure");
    caption = "This is the UML Diagram ";

    // Set Diagram orientation to portrait; default
    // is landscape
    diagram_orientation_scale_set(PortraitFit);

    // Use OMS query to retrieve from the repository
    // all UML diagrams whose names fit the match  Selects
    // pattern "Email_*"                           diagrams
    for_each_in_select("file[UmlClassDiagram &&    from
        name $ 'Email_*']", file_var)              repository
    {
        file_print(file_var, caption + file_var.name);
    }                                          Prints the
}                                              diagrams
```

This script produces a MIF file. When you open the document in FrameMaker, a typical page resembles Figure 19.

Figure 19: Sample Document Page



Including Tables and Diagrams in a Microsoft Word or HTML Document

To modify the example script for Microsoft Word or HTML, substitute the arguments to the paragraph and diagram_caption_paragraph_format_set functions with values appropriate to the publishing tool.

Alternatively, declare all variables and corresponding publishing tool values at one time:

1. Declare variables and assign values for the appropriate publishing tool.

   The following example defines a variable and provides a value to the diagram_caption_paragraph_format_set function:

   ```
   const string FigureFormat;

   if (target() == rtf)
      FigureFormat="DiagramTitle";

   diagram_caption_paragraph_format_set(FigureFormat)
   ```

   where DiagramTitle is a paragraph tag defined in a format file.

2. Run the script with the appropriate options for the publishing tool, such as:

   `qrp script.qrl -t rtf - f format.rtf -o output.rtf`

   where *script.qrl* is the script name, *rtf* is the target, *format.rtf* is the format file, and *output.rtf* is the output file.

Modifying Print Option Values

This section describes the functions that enable you to specify values for:

- Alignment, orientation, placement, position, size, and scaling of printed diagrams, tables, and diagram and table captions
- Character and paragraph formats in diagram and table captions, diagram labels, and table cells and headings
- Hyperlink destinations for diagrams and tables in HTML-generated documents
- Table rulings and shadings

There are two types of print option functions:

- Functions that change or reset the default print option values

  These functions are described in this section.
- Functions that override or reset print option values set in a named setting

  These functions are described in "Overriding Named Settings" on page 5-76.

Print option functions take the form:

```
diagram|table_<format_option>_set(<format_value>);
```

For example:

```
diagram_position_from_top_set(4.5);
table_caption_paragraph_format_set("TableTitle");
table_row_range_set("1-3");
```

These functions are used in conjunction with the file_print function to apply a printing option to the table or diagram being printed. For example:

```
// Use diagram_orientation_scale_set function to define how
// diag_var is printed
diagram_orientation_scale_set(PortraitFit);
file_print(diag_var, "");
```

There is a one-to-one correspondence between many of these options and those that can be implemented through the **Print** dialog box in the StP editors. For more information, see *Fundamentals of StP*, or "Using Named Settings" on page 5-70.

Enumeration Types for Print Options

Several of the printing option functions take enumeration types as their arguments.

An enumeration type is a data type for which a set of all possible values is defined. The enumeration types for printing options are listed in Table 14.

Table 14: Enumeration Types for Printing Options

Type Name	Permitted Values
alignment_enum	Left, Center, Right
caption_orientation_enum	Portrait, Landscape, AsDiagram
orientation_scale_enum	BestFit, LandscapeFit, PortraitFit, LandscapeScale, PortraitScale
placement_enum	Above, Below
position_enum	Centered, Specified
table_orientation_enum	PortraitTable, LandscapeTable (Supported with FrameMaker and RTF.)
target_enum	Ascii, Html, Framemaker, RTF
units_enum	Inches, Cm

Enumeration values must be used exactly as they are defined—that is, the case must match. For example, the enumeration type alignment_enum has three possible values: Left, Right, and Center. Supplying any value other than Left, Right, or Center to a function that takes an alignment_enum as an argument results in an error.

Allowable values for placement_enum can include Top and Bottom, for the sake of backward compatibility with older versions of StP.

Other examples are:

```
// use orientation_scale_enum value
diagram_orientation_scale_set(PortraitFit);

// use placement_enum value
table_caption_placement_set(Below);
```

Functions for Modifying Print Options

The functions for changing or resetting the print options for tables and diagrams include the _set functions, and their _reset counterparts. The _set functions enable you to set options for printing tables and diagrams

that differ from the default values. The _reset functions cancel the corresponding _set function or all _set functions.

Table 15 lists the _set functions for diagrams; Table 16 lists them for tables. For allowable values for _enum types, see Table 14 on page 5-51.

The phrase “current units” means the current value set by the file_print_units_set function.

Table 15: Diagram Printing Option _set Functions

Function	Return Type	System Default	Description
diagram_caption_alignment_set (alignment_enum <value>)	alignment_ enum	Center	Sets the horizontal adjustment of the caption.
diagram_caption_orientation_set (caption_orientation_enum <value>)	caption_ orientation _enum	AsDiagram	Sets the orientation of the caption on the page. Permitted values are Portrait, Landscape, and AsDiagram.
diagram_caption_paragraph_ format_set (string <value>)	string	"StpDefault"	Sets the paragraph format of the caption.
diagram_caption_placement_set (placement_enum <value>)	placement_ enum	Below (For RTF, caption placement must always be Above)	Sets the vertical placement of the caption.
diagram_display_mark_no_print_set (set <value>)	set	NULL	Specifies that a user-defined set of display marks is not to be displayed on the printed diagram. See note after this table regarding controlling the printing of display marks.
diagram_display_mark_character_ format_set (string <value>)	string	"StpDefault"	Sets the character format of display marks.
diagram_label_character_format_set (string <value>)	string	"StpDefault"	Sets the character format of the diagram label.

Table 15: Diagram Printing Option _set Functions (Continued)

Function	Return Type	System Default	Description
diagram_frame_height_set (float <value>)	float	504 pts (7 inches)	Sets the height of the diagram frame in current units.
diagram_frame_width_set (float <value>)	float	324 pts (4.5 inches)	Sets the width of the diagram frame in current units.
diagram_mark_set (string mark_label)	string	"" (no mark set)	Sets the diagram as a mark, that is, as the destination of an HTML hyperlink.
diagram_orientation_scale_set (orientation_scale_enum <value>)	orientation _scale_ enum	BestFit	Sets the orientation and scaling mechanism of the diagram on the page. Permitted values are Portrait-Scale, LandscapeScale, PortraitFit, LandscapeFit, and BestFit.
diagram_position_from_left_set (float <value>)	float	0	Sets the position of the diagram, in current units, from the left of the margin. Only applicable when the Orientation / Scale option is set to Portrait or Landscape, or the diagram position is not Centered.
diagram_position_from_top_set (float <value>)	float	0	Sets the position of the diagram, in current units, from the top margin. Only applicable when the Orientation / Scale option is set to Portrait or Landscape, or the diagram position is not Centered.
diagram_position_set (position_enum <value>)	position_ enum	Centered	Sets the position of the diagram relative to the margins. Not applicable the Orientation / Scale option is set to one of the "Fit" values.

Table 15: Diagram Printing Option _set Functions (Continued)

Function	Return Type	System Default	Description
diagram_show_caption_set (boolean <value>)	boolean	True	Specifies whether diagram caption is shown.
diagram_scale_percent_set (float <value>)	float	100.0	Sets diagram size in percent of diagram shown in editor at 100%. Not applicable if Orientation / Scale option is set to one of the "Fit" values.

Note: There is a diagram editor rule that suppresses the appearance of specific display marks on diagrams. This is the NoPrintDisplayMarks rule, which holds for diagrams printed with **dprint** or **qrp**. It does not hold for diagrams printed from an editor, which are printed as shown in the editor. See *Customizing StP* for information on editor rules.

Table 16 lists the functions that set print options for tables. For allowable values for _enum types, see Table 14 on page 5-51. The phrase "current units" means the current value set by the file_print_units_set function.

Table 16: Table Printing Option _set Functions

Function	Return Type	System Default	Description
table_caption_alignment_set (alignment_enum <value>)	alignment_ enum	Center	Sets the horizontal placement of the table caption.
table_caption_paragraph_format_set (string <value>)	string	"StpDefault"	Sets the paragraph format of the table caption.
table_caption_placement_set (placement_enum <value>)	placement_ enum	Above	Sets the vertical placement of the table caption.
table_show_caption_set (boolean <value>)	boolean	True	If True, table caption is shown.
table_cell_bold_paragraph_ format_set (string <value>)	string	"StpDefault"	Sets the paragraph format for bold text in table cells.

Table 16: Table Printing Option _set Functions (Continued)

Function	Return Type	System Default	Description
table_cell_paragraph_format_set (string <value>)	string	"StpDefault"	Sets the paragraph format for text in table cells.
table_column_range_set (string <value>)	string	"" (all columns)	Sets which table columns to print. Specified as a comma-separated list of intervals that are ordered and do not overlap, for example "1-3, 5, 8-*".
table_header_row_range_set (string <value>)	string	"" (no rows)	Sets which rows are repeated at the top of every page if a table spans more than one page. Specified in same way as table_column_range_set.
table_height_set (float <value>)	float	504 pts (7 inches)	Sets the height of the table in current units.
table_mark_set(string mark_label)	string	"" (no mark set)	Sets the table as a mark, that is, as the destination of an HTML hyperlink.
table_row_range_set (string <value>)	string	"" (all rows)	Sets which rows are printed in the output. Specified in same way as table_column_range_ set.
table_maximum_vertical_span_set (int <value>)	int	5	Sets the maximum number of rows a table cell can span.
table_orientation_set (table_orientation_enum <value>)	table_ orientation _enum	Portrait Table	MIF or RTF only. Sets whether the table is created in a portrait or landscape format.
table_print_column_indices_set (boolean <value>)	boolean	True	Sets whether columns of the table are numbered in the output file.
table_print_row_indices_set (boolean <value>)	boolean	True	Sets whether rows of the table are numbered in the output file.

Table 16: Table Printing Option _set Functions (Continued)

Function	Return Type	System Default	Description
table_broken_ruling_set (string <value>)	string	"Broken"	Sets the "broken" ruling to a different ruling type.
table_double_ruling_set (string <value>)	string	"Double"	Sets the "double" ruling to a different ruling type.
table_medium_ruling_set (string <value>)	string	"Medium"	Sets the "medium" ruling to a different ruling type.
table_thick_ruling_set (string <value>)	string	"Thick"	Sets the "thick" ruling to a different ruling type.
table_thin_ruling_set (string <value>)	string	"Thin"	Sets the "thin" ruling to a different ruling type.
table_shade1_color_set (int <value>)	int	4	Sets color for table shadings. See the target publishing tool documentation for the meaning of these integer values.
table_shade2_color_set (int <value>)		4	
table_shade3_color_set (int <value>)		4	
table_shade1_pattern_set (int <value>)	int	4	Sets pattern for table shadings. See the target publishing tool documentation for the meaning of these integer values.
table_shade2_pattern_set (int <value>)		3	
table_shade3_pattern_set (int <value>)		2	
table_width_set (float <value>)	float	432 pts (6 inches)	Sets the width of the table in current units.

All of the _set functions in Table 15 and Table 16 have a corresponding _reset function. The _reset functions restore the default values of options that have been modified with _set functions.

The _reset functions have the form:

```
void diagram_<format_option>_reset()
void table_<format_option>_reset()
```

where <format_option> stands for the option that completes the function. For example:

```
diagram_frame_height_reset()
table_print_column_reset()
```

All _reset functions have a return type of void.

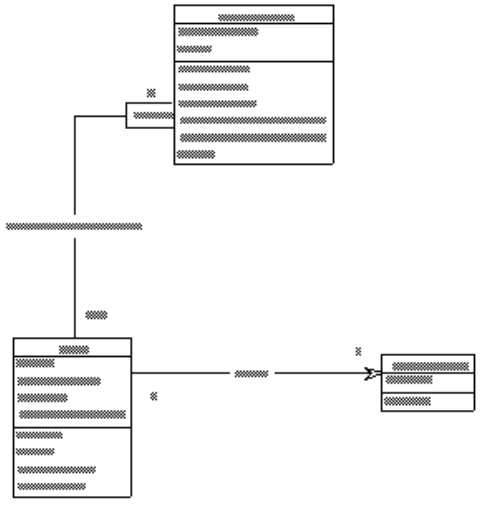
Two other _reset functions exist, not shown in Table 15 and Table 16, that reset all values to their defaults for diagrams and tables respectively:

```
diagram_reset_all()
table_reset_all()
```

Using the _set and _reset Functions

The script in Figure 21 on page 5-58 illustrates how the _set and _reset functions work together. This script selects the same diagrams for printing as the script in Figure 18 on page 5-47. The script in Figure 18 used diagram_orientation_scale_set to print the diagrams in an orientation other than the default (that is, in PortraitFit rather than BestFit). Some of the diagrams picked up by that script do not fit well if printed with PortraitFit orientation. For example, the labels in the diagram in Figure 20 are extremely small.

Figure 20: Diagram Labels are Too Small



To correct the problem with the labels, the script in Figure 21 uses two tactics: enlarging the width of the frame, and letting StP select the best fit. The script selects the same diagrams, but uses diagram_frame_width_set to make the frames wider, then resets the width to the default, selects the diagrams again, and uses diagram_orientation_scale_reset to restore the default orientation, so that the diagrams are formatted in BestFit mode.

Figure 21: Sample Script: Using _set and _reset Functions

```
void
main()
{
   file file_var;
   string caption;

   // Set diagram frame width to 5.75
   diagram_frame_width_set(5.75);
```

```
    // Apply paragraph format "Figure" to diagram
    // caption
    diagram_caption_paragraph_format_set("Figure");
    caption = "This is Diagram ";

    // Set Diagram orientation to PortraitFit; default
    // is BestFit
    diagram_orientation_scale_set(PortraitFit);

    // Print all diagrams with names of "Email_*" pattern
    // using PortraitFit orientation
    for_each_in_select("file[UmlClassDiagram && +
           name $ 'Email_*']", file_var)
           file_print(file_var, caption + file_var.name);

    // Reset diagram frame width and orientation to
    // the defaults
    diagram_frame_width_reset();
    diagram_orientation_scale_reset();

    // Print all diagrams as above, but with default
    // frame widths and orientation
    for_each_in_select("file[UmlClassDiagram && +
           name $ 'Email_*']", file_var)
           file_print(file_var, caption + file_var.name);
}
```

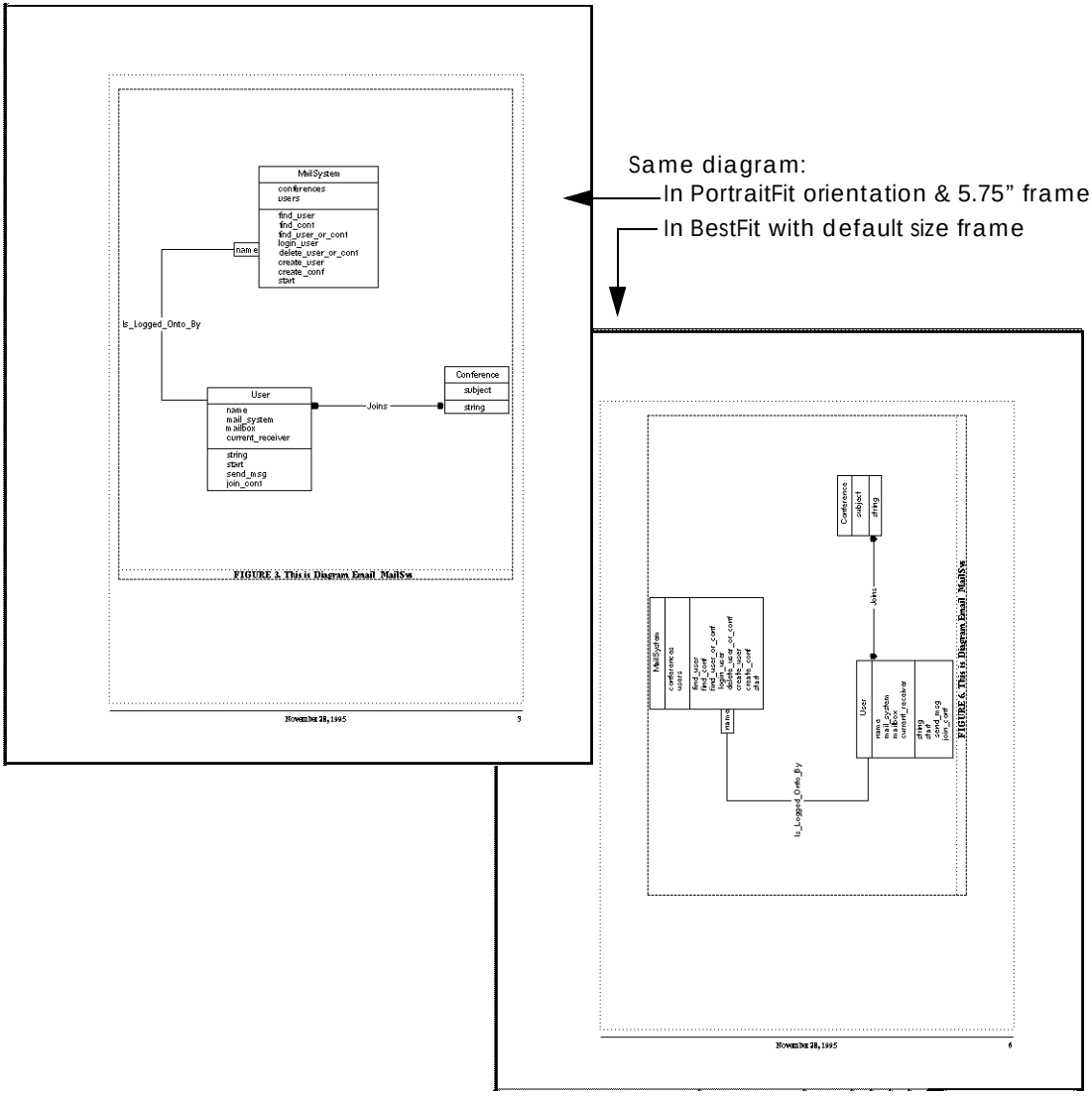
You run this script by typing:

```
qrp script.qrl -t mif -f report.mif -o uml_diag.mif
```

where *script.qrl* is the script name, mif is the target, *report.mif* is the format file, and *uml_diag.mif* is the output file.

This script produces a MIF file. Figure 22 shows two pages from the resulting document: one with a diagram from Figure 20 printed in portrait orientation with a frame width 5.75", and the other with the same diagram with its default size frame printed in landscape.

Figure 22: Diagrams Showing Different Print Option Values



To modify the script in Figure 21 on page 5-58 for Microsoft Word or HTML, see "Including Tables and Diagrams in a Microsoft Word or HTML Document" on page 5-49.

Creating Hypertext Links in Diagrams and Tables

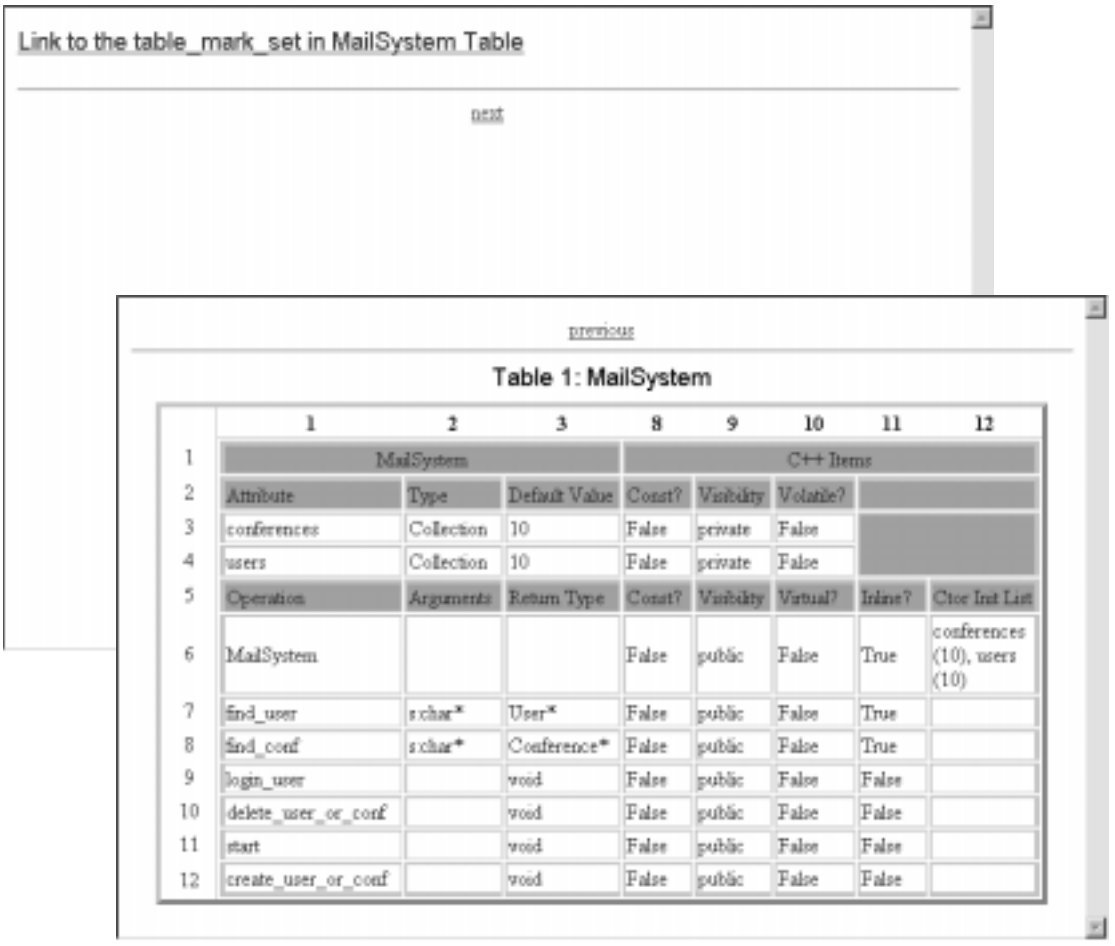
For HTML-generated documents, you define explicit hyperlinks to diagrams and tables using the diagram_mark_set or table_mark_set function, respectively.

The following example script designates the table, MailSystem, as the destination of a hyperlink mark:

```
void
main()
{
    file f;

    print_link("Link to the table_mark_set in MailSystem
Table", "mark label 1");

    document("", "");

    table_mark_set("mark label 1");
    table_cell_paragraph_format_set("P.class1");
    table_cell_bold_paragraph_format_set("P.class2");
    f = find_by_query("file[UmlClassTable &&
name='MailSystem']");
    file_print(f, "Table 1: MailSystem");
}
```

The script produces two HTML files, as shown in Figure 23.

Figure 23: Diagrams Showing an HTML Table Link



	1	2	3	8	9	10	11	12
1	MailSystem			C++ Items				
2	Attribute	Type	Default Value	Const?	Visibility	Volatile?		
3	conferences	Collection	10	False	private	False		
4	users	Collection	10	False	private	False		
5	Operation	Arguments	Return Type	Const?	Visibility	Virtual?	Inline?	Ctor Init List
6	MailSystem			False	public	False	True	conferences (10), users (10)
7	find_user	s:char*	User*	False	public	False	True	
8	find_conf	s:char*	Conference*	False	public	False	True	
9	login_user		void	False	public	False	False	
10	delete_user_or_conf		void	False	public	False	False	
11	start		void	False	public	False	False	
12	create_user_or_conf		void	False	public	False	False	

Customizing Shading in a Table

This section describes how to set non-default values for shading and color in tables formatted with a QRL script.

Aonix provides this information based on the latest release of the appropriate software. This information is subject to change. For complete

information on color and shading, see your FrameMaker, Microsoft Word, or HTML and CSS documentation.

StP table editors let you specify three different shadings for individual cells or groups of cells. These are available using the **Edit > Format Current Selection** command. They are designated Shade 1, Shade 2, and Shade 3. You can determine how each of these shades will be displayed or printed in tables you format with a QRL script.

For StP table printing, each shade consists of a color and a pattern. Each designated shade (Shade 1, Shade 2, and Shade 3) consists of the following two functions:

- table_shade[1 | 2 | 3]_color_set
- table_shade[1 | 2 | 3]_pattern_set

For example, Shade 1 composes the functions table_shade1_color_set and table_shade1_pattern_set.

Specifying Color Values

FrameMaker, HTML, and Microsoft Word allow you to specify a color using a corresponding integer. You use a corresponding integer with the table_shade_color_set function:

- For FrameMaker, see Table 17
- For Microsoft Word, see Table 18 on page 5-64
- For HTML, see Table 19 on page 5-65

Table 17: FrameMaker Shade_Color Values

int Value	Color
0	Black
1	White
2	Red
3	Green
4	Blue

Table 17: FrameMaker Shade_Color Values (Continued)

int Value	Color
5	Cyan
6	Magenta
7	Yellow

Table 18: RTF Shade_Color Values

int Value	Color
0	default for document
1	Black
2	Blue
3	Cyan
4	Green
5	Magenta
6	Red
7	Yellow
8	White
9	Dark Blue
10	Dark Cyan
11	Dark Green
12	Dark Magenta
13	Dark Red
14	Dark Yellow
15	Dark Gray
16	Light Gray

Table 19: HTML Shade_Color Values

int Value	Color
0	Black
1	Silver
2	Gray
3	White
4	Light Blue
5	Maroon
6	Red
7	Purple
8	Fuchsia
9	Green
10	Lime
11	Olive
12	Yellow
13	Navy
14	Blue
15	Teal
16	Aqua

The default values for all the table_shade_pattern_set and table_shade_color_set functions are given in Table 16 on page 5-54.

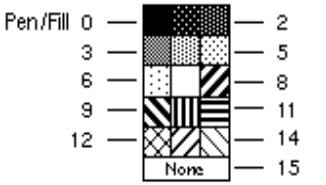
Note: The colors used for integer values in QRL scripts depend on the values in the color table in the format files used. For example, by default, Microsoft Word uses the same values as those in Table 18 when it writes a color table to an RTF file. Unless you customize the color table in your format file, the integer-to-color correspondence will be as shown above.

Specifying Pattern Values

You also use integers to specify patterns. You use a corresponding integer with the table_shade_pattern_set function:

- For FrameMaker, see Figure 24
- For Microsoft Word, see Table 20 on page 5-66
- For HTML, see Table 21 on page 5-67

Figure 24: FrameMaker Shade_Pattern Values



Pen/Fill Patterns in Tools palette

Table 20: Microsoft Word Shade_Pattern Values

int Value	Pattern
0-15	Progressively darker, evenly distributed shading
16	Dark horizontal
17	Dark vertical
18	Dark forward diagonal
19	Dark backward diagonal
20	Dark cross
21	Dark diagonal cross
22	Horizontal
23	Vertical
24	Forward diagonal

Table 20: Microsoft Word Shade_Pattern Values (Continued)

int Value	Pattern
25	Backward diagonal
26	Cross
27	Diagonal cross

Table 21: HTML Shade_Pattern Values

int Value	Pattern
any value except 2 or 3	Same as chosen color
2	Darker variation of chosen color
3	Darkest variation of chosen color

Example Script with Modified Shading

The example script in Figure 25 produces a MIF file and prints a table twice: first with the default values for Shade 2, and then changes the setting of Shade 2 to a pattern value of 6 (which is lighter), leaving the color value as the default.

The script in Figure 25 also removes the indices for both columns and rows and creates a caption for the table, which it places below the table.

Figure 25: Sample Script: Formatting a Table

```
void
main()
{
    file file_var;
    string caption;

    // Apply paragraph format Figure to table
    // caption and give it some text
    table_caption_paragraph_format_set("Figure");
    caption = "This is the Class Table for the Class ";
```

```
    // Place the caption below the table
    table_caption_placement_set(Below);

    // Remove all indices
    table_print_column_indices_set(False);
    table_print_row_indices_set(False);

    // Use OMS query to retrieve from the repository
    // the UML Class Table for the Message class
    for_each_in_select("file[UmlClassTable && name ==
           'Message']", file_var)
    {
       file_print(file_var, caption + file_var.name);
    }

    // Set shading for Shade2 to a lighter color,
    // force a page break, and print the same table
    table_shade2_pattern_set(6);
    page();
    for_each_in_select("file[UmlClassTable && name ==
           'Message']", file_var)
    {
       file_print(file_var, caption + file_var.name);
    }
}
```

Note: Both for_each_in_select statements can be replaced with find_by_query statements, since only one table is sought. Using for_each_in_select makes the script more easily customizable, if you should want to search for multiple tables.

To modify the example script for RTF or HTML, substitute the arguments to the paragraph and table_caption_paragraph_format_set functions with values appropriate to the publishing tool.

Alternatively, declare all variables and corresponding publishing tool values at one time:

1. Declare variables and assign values for the appropriate publishing tool.

   The following example defines a variable and provides a value to the table_caption_paragraph_format_set function:

   ```
   const string TableFormat;
   ```

```
if (target() == rtf)
   TableFormat="TableTitle";

table_caption_paragraph_format_set(TableFormat)
```

where TableTitle is a paragraph tag defined in an RTF format file.

2. Change the table_shade2_pattern_set function:
   For Microsoft Word shade values, see Table 18 on page 5-64.

   For HTML shade values, see Table 21 on page 5-67.

3. Run the script with the appropriate options for the publishing tool, such as:

   `qrp script.qrl -t rtf -f format.rtf -o output.rtf`

   where *script.qrl* is the script name, *rtf* is the target, *format.rtf* is the format file, and *output.rtf* is the output file.

RTF Limitations

StP diagrams and tables may contain features that are not supported by RTF and Microsoft Word. These features include:

- If a cell spans more than one row, its label is duplicated in each row in the generated table.
- A split cell is simulated as closely as possible in the generated table.
- Header rows in the body of the table must begin with Row 1 and be contiguous.
- Diagram captions are generated in the same orientation as the diagram itself, and always appear above the diagram.
- To view diagrams in Microsoft Word, you must use Page Layout view (select **Page Layout** from the Microsoft Word **View** menu).

HTML Limitations

The following limitations apply to HTML diagram and table formatting:

- Printing tables in landscape format is not currently supported. Tables must be printed in portrait format.
- Generated diagrams are currently in RTF format. The generated HTML document contains a link to the appropriate RTF file.

- Due to limitations in Netscape Navigator, borders on empty table cells may not appear.

Using Named Settings

Print settings can be modified for a particular table or diagram and then explicitly associated with that table or diagram as a named setting. This section explains how you can use this named setting in a script.

You can use named setting naming conventions to determine when a named setting is used. Table 22 shows the various options.

Table 22: Named Setting Names and QRL

Named Setting Name	When to Use
<unique_ name>	When you want to use this setting for this diagram in a script (as the <named_setting> argument to file_print)
<script_name>	When you want this setting used for this diagram throughout the <script_name> script (no <named_setting> argument to file_print)
qrs	When you want this setting used for this diagram in any QRL script (no <named_setting> argument to file_print)

For example, assume that for a diagram named *my_diagram* you create three named settings (using the **Print** dialog box from an editor) with the following names:

- *setting_1*
- *the_script*
- *qrs*

These named settings are stored in a file associated with the *my_diagram* diagram.

Examples in the following sections will illustrate how these settings can be used.

Explicit Use of Named Settings

To print *my_diagram* using a specific setting, you use the name of the desired setting as an argument to file_print. For example:

```
file_print(file_var,"", "setting_1");
```

The rules of precedence are:

1. **qrp** first looks for a named setting with this name in the named setting file associated with the table or diagram being printed.
2. If a setting by this name does not exist, the behavior reverts to the implicit case, described next.

Implicit Use of Named Settings

Implicit use of named settings requires that a setting exists with the same name as the script name, or the name "qrs." Implicit use of named settings is in effect when you call file_print without specifying a <named_setting> argument.

The rules of precedence are:

1. **qrp** first looks for a setting associated with the object with the same name as the script being executed.

   qrp disregards the script's extension; for a script named *the_script.qrl*, **qrp** looks for the *the_script* setting name.
2. If that is not found, **qrp** then looks for a setting associated with the object with the name "qrs."
3. If neither of these exists, **qrp** uses whatever options are specified in the script.
4. If there are no options set in the script, **qrp** uses default print option values.

For example usage, see "Named Setting Example" on page 5-74.

When you use a script to print a table or diagram that has a named setting associated with it, the following rules apply:

- You can use file_print_named_settings_use() to specify whether named settings are available for use or not.

  For example, calling file_print_named_settings_use (False) turns off all named settings for all diagrams and tables to the end of the script or until you call file_print_named_settings_use (True).
- You can only apply a named setting to the table or diagram file with which it is associated.
- When you use a named setting to print a table or diagram, options that have no corresponding **Print** command setting are merged with the named settings.

Rules of Precedence for Print Options

qrp applies print option values in the following order of precedence, from lowest to highest:

1. Default values
2. Values supplied by _set functions
3. Values supplied by named settings
4. Values supplied by _override functions

   You can override a particular value specified in a named setting with its corresponding _override function. These are described in "Overriding Named Settings" on page 5-76.

Table 23 gives examples illustrating these rules.

Table 23: Printing With and Without Named Settings

If you are...	You can do one of the following...
Printing a table or diagram that has no named setting associated with it	Print the table or diagram without changing any of the default printing options (Rule 1).
	Use one of the **_set** functions to change default settings (Rule 2).

Table 23: Printing With and Without Named Settings (Continued)

If you are...	You can do one of the following...
Printing a table or diagram that has a named setting associated with it	Print the table or diagram without using the named setting (Rule 1). (There must be no named setting *qrs* or the same name as the script.)
	Implicitly or explicitly apply the named setting to the table or file being printed (Rule 3).
	Turn off any active named settings with file_print_named_settings_use (False).
	Turn named settings back on with file_print_named_settings_use (True).
	Use one of the _override functions to override the corresponding print option (Rule 4).

To better understand how named settings are used in printing tables and diagrams, consider the following scenario: you have included a query in a script that retrieves all UML class diagrams in the system:

```
for_each_in_select("file[UmlClassDiagram]", diag_var)
```

Assume there are four diagrams in the system. Three of the diagrams have two named settings associated with each of them: *setting_1* and *setting_2*. The fourth diagram has no named setting. You want to use *setting_2* to print the diagrams. You can do one of the following:

- Use the following statement in the script used to print the diagrams:

  ```
  file_print(diag_var,"","setting_2");
  ```

- Use file_print without a <named_setting> argument and:

  Name the script *setting_2.qrl* or

  Change the name of the setting *setting_2* to *qrs*.

No matter which method you use, the script produces a document with the first three diagrams printed with *setting_2* values, but the fourth

diagram is printed with the default print settings. This is because named settings can only apply to the diagram or table they are associated with.

In addition, even though the other three diagrams were associated with a named setting named *setting_2*, the actual values of the settings may differ. Each diagram will be printed with its own version of *setting_2*.

Named Setting Example

The script in Figure 26 demonstrates how and when settings are applied in a script. The table used in the example is an StP/UML class table, but the principles apply to any table.

This example applies four named settings:

- *setting_1*
- *setting_2*
- *namedset*
- *qrs*

The script also includes two _set functions:

```
table_caption_paragraph_format_set()
table_width_set()
```

The first _set function specifies a format for the table caption; this is not affected when a named setting is applied, because the table caption option can not be saved in a named setting. However, the function table_width_set() specifies a setting that can be specified in a named setting; when both are specified for a single file_print statement, the value specified in the named setting takes precedence.

Figure 26: Sample Script: Using Named Settings

```
external string TABLE = "";

void
main()
{
   file file_var;
   string caption = "This table uses ";
```

```
    for_each_in_select("file[UmlClassTable && " +
           "name == '${TABLE}']", file_var)
    {
       // Print Table 1 using implicitly applied settings.
       // Which ones are used depends on the name of
       // this script (either "qrs" or "namedset").
       file_print(file_var, caption + "an implicit
              setting");

       // Break the page and print Table 2 using saved
       // setting "setting_1"
       page();
       file_print(file_var, caption + "setting_1",
              "setting_1");

       // Set caption to use TableTitle format defined in
       // report.mif format file
       table_caption_paragraph_format_set("TableTitle");

       // Turn off all named settings
       file_print_named_settings_use(False);

       // Break page and print Table 3 without saved
       // setting. However, table_caption format
       // specified previously is still in effect
       page();
       file_print(file_var, caption + "the default" +
              "setting");

       // Use _set function to set table width to 5.5
       table_width_set(5.5);

       // Attempt to apply a named setting (Table 4)
       // without turning them back on
       page();
       file_print(file_var, caption + "a set width",
              "setting_2");

       // Turn on named settings and reissue last call
       // (print Table 5)
       file_print_named_settings_use(True);
       page();
       file_print(file_var, caption + "print", "print");
    }
}
```

When you run this script by typing:

```
qrp namedset.qrl -x TABLE Circle -t mif -f report.mif
       -o namedset.mif
```

the output is a MIF document consisting of five instances of the Circle class table:

- Table 1 is displayed with the *namedset* setting values.

  Had the script been named anything but *namedset.qrl*, this table would have used values from the *qrs* named setting for this table.
- Table 2 is displayed with *setting_1* setting values.

  The *setting_1* setting specifies landscape orientation. **qrp** only prints tables in portrait orientation, and this table runs off the page.
- Table 3 is displayed with the default table setting values; the caption is formatted with the TableTitle format.
- Table 4 also uses the default setting, but with a width of 5.5 inches.

  Despite using the named setting argument, system default values were used because named settings were still turned off.
- Table 5 is displayed with *setting_2* settings.

  The table_width_set value was ignored; the *setting_2* setting is of higher precedence. However, the caption is in TableTitle format.

Overriding Named Settings

As demonstrated in the last example ("Named Setting Example" on page 5-74), _set functions cannot override values that are set with named settings. QRL provides a set of functions that can override some named setting values. These are the _override functions.

Just as QRL provides the _reset functions to undo the effects of the _set functions, QRL also provides a set of _override_clear functions to undo the effects of the _override functions.

Table 24 lists the _override functions that apply to diagram and table print settings. Compare with the corresponding _set functions in Table 13 on page 5-45, Table 15 on page 5-52, and Table 16 on page 5-54. For a list of possible values for the _enum return types, see Table 14 on page 5-51.

Table 24: Print Option _override Functions

Function	Return Type
diagram_display_mark_no_print_override (set <value>)	set
diagram_mark_override(string mark_label)	string
diagram_orientation_scale_override (orientation_scale_enum <value>)	orientation_scale_ enum
diagram_position_override (position_enum <value>)	position_enum
diagram_frame_height_override (float <value>)	float
diagram_frame_width_override (float <value>)	float
diagram_position_from_top_override (float <value>)	float
diagram_position_from_left_override (float <value>)	float
diagram_show_caption_override (boolean <value>)	boolean
diagram_scale_percent_override (float <value>)	float
table_column_range_override (string <value>)	string
table_header_row_range_override (string <value>)	string
table_height_override (float <value>)	float
table_mark_override(string mark_label)	string
table_maximum_vertical_span_override (int<value>)	int
table_orientation_override (table_orientation_enum <value>)	table_orientation_ enum
table_print_column_indices_override (boolean <value>)	boolean

Table 24: Print Option _override Functions (Continued)

Function	Return Type
table_print_row_indices_override (boolean <value>)	boolean
table_row_range_override (string <value>)	string
table_show_caption_override (boolean <value>)	boolean

All of the _override functions in Table 24 have a corresponding _override_clear function. The _override_clear functions have the form:

```
void diagram_<format_option>_override_clear()
void table_<format_option>_override_clear()
```

where <format_option>stands for the option text that completes the function. For example:

```
diagram_frame_height_override_clear()
table_print_column_indices_override_clear()
```

All _override_clear functions have a return type of void.

Two other _override_clear functions exist, not shown inTable 24, that reset all values to their named setting values for diagrams and tables respectively:

```
diagram_override_clear_all()
table_override_clear_all()
```

Extended Example

This section provides a script that creates a document with:

- A text file, *uc_summary.txt*, that is imported into the document by the read_file function
- A diagram
- Information about the diagram, all of the nodes (a PDM type) on it, and all of the notes associated with each node
- An external variable, DIAGRAM, that specifies the name of the diagram

The following script is run with a user-defined FrameMaker format file, *my_formats.mif,* which contains the following paragraph formats:

- Title
- Summary—like Title, but with a smaller type
- Body
- Subhead1—an unnumbered heading
- Subhead2—like Subhead1, but with a smaller type

You run the script by typing:

```
qrp script.qrl -x DIAGRAM UseCases -t mif
    -f ./my_formats.mif -o extended_report.mif
```

where *script.qrl* is the name of the script, *UseCases* is the name of the Use Case diagram for which the report is being generated, “mif” is the target, *./my_formats.mif* is the name and path of the format file (it is in the current directory), and *extended_report.mif* is the name of the output document.

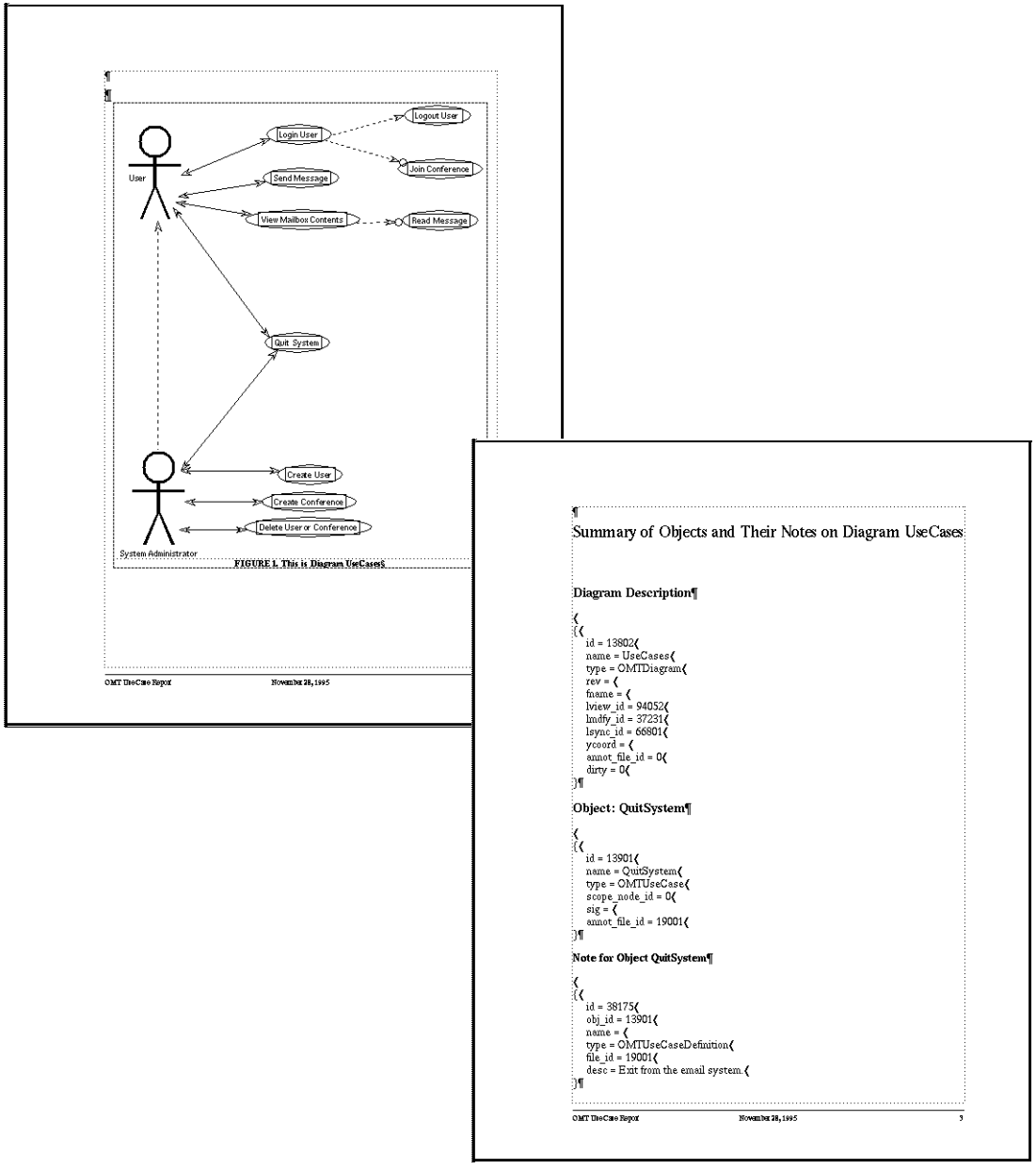
```
external string DIAGRAM = "";

void
main()
{
    file file_var;
    string caption;
    string str_file = "";
    node node_var;
    note note_var;
```

```
// Apply "Title" paragraph format
paragraph("Title");
print("UML Use Case Report");

// Apply "Body paragraph format
paragraph("Body");

// Read in text to be included in document
str_file = read_file("uc_summary.txt");
print(str_file);

// Start a new page
page();

// Apply "Figure" paragraph format to diagram caption
diagram_caption_paragraph_format_set("Figure");

// Assign string "This is Diagram: " plus the value
// of DIAGRAM to the variable caption
caption = "This is Diagram " + DIAGRAM;

// Make sure the diagram fits the page
diagram_frame_width_set(5.75);

// Set diagram orientation to PortraitFit
diagram_orientation_scale_set(PortraitFit);

// Retrieve diagram DIAGRAM from the repository
for_each_in_select("file[UmlUseCaseDiagram && " +
   "name == '${DIAGRAM}']", file_var)
{
   // Print the diagram and a caption
   file_print(file_var, caption);

   // Start a new page
   page();

   // Apply "Summary" paragraph format to title
   paragraph("Summary");
   print("Summary of Objects and Their " +
      "Notes on Diagram " + DIAGRAM);

   // Apply "Subhead1" paragraph format to create
   // unnumbered head
   paragraph("Subhead1");
   print("Diagram Description");

   // Apply "Body" paragraph format and print
   // print the description associated with
```

```
        // the diagram DIAGRAM
        paragraph("Body");
        print(file_var);

        // Retrieve all of the nodes associated with
        // DIAGRAM from the repository
        for_each_in_select("node[node_refs[file" + "
            [UmlUseCaseDiagram && name == " +
            "'${DIAGRAM}']]]",
             node_var)
        {
            // Apply "Subhead1" paragraph format and
            // print the name of the node
            paragraph("Subhead1");
            print("Object: " + node_var.name);

            // Apply "Body" paragraph tag; print
            // information about the node
            paragraph("Body");
            print(node_var);

            // Retrieve notes associated with current node
            // from the repository
            for_each_in_select("note[obj_id == " +
                "${node_var.id}]", note_var)
            {
                // Apply "Subhead2" paragraph format; print
                // the name of the node
                paragraph("Subhead2");
                print("Note for Object " +
                       node_var.name);

                // Apply "Body" paragraph format and
                // print the note's information
                paragraph("Body");
                print(note_var);
            }
        }
    }
}
```

Sample pages of the output file are shown in Figure 27 on page 5-82.

Figure 27: Extended Example Sample Output Pages



A Built-in Functions

This chapter provides an alphabetical list of all the built-in functions supported by QRL. Each function is identified by type. Descriptions and examples of these function types are provided elsewhere in this manual, as shown in Table 1.

Table 1: Function Type and Description Reference

For this function type...	Refer to this section...
Enumeration type	“Enumeration Types” on page 3-36
List type	“The list Type” on page 3-40
Set type	“The set Type” on page 3-50
Graph type	“The graph Type” on page 3-54
String manipulation	“String Manipulation Functions” on page 3-58
System	“Functions for Returning System Information” on page 3-64
Repository	“Administrative Functions” on page 3-67
System repository	
QRL debugging	“QRL Debugging Features” on page 3-91
OMS Interface	“OMS Interface Functions” on page 4-2
Formatting & printing text	“Built-in Text Formatting and Printing Functions” on page 5-25

Table 1: Function Type and Description Reference (Continued)

For this function type...	Refer to this section...
Printing tables & diagrams	"Printing Tables and Diagrams" on page 5-43
Table options	
Diagram options	

Syntax Summary

Function Declaration

The syntax for function declaration is:

```
<return_type>
<function_name>([<type> <param>] [,<type> <param>...])
{
   statements
   [return [<expression>];]
}
```

A return statement is required for all functions, except for functions declared as void.

Function Call

A function call has the following format:

```
<function_name>([<argument>] [,<argument>...]);
```

Each item in the argument list corresponds positionally to a parameter in the function definition. For example, the function:

```
int func1(string var1, int var2)
```

is called from another function as follows:

```
func1("this is the string value for var1", 5);
```

where `"this is the string value for var1"` is the argument for `var1` and `5` is the argument for `var2`.

Toggle Functions

This section describes two groups of function pairs that toggle print option settings. Since the effect of the “toggle off” functions is predictable, they are not listed in the function list in Table 2 on page A-5.

_set and _reset Functions

All diagram_<format_option>_set and table_<format_option>_set functions have a corresponding _reset function. _reset functions restore the default values of options that have been modified with _set functions.

The _reset functions have the form:

```
void diagram_<format_option>_reset()
void table_<format_option>_reset()
```

where <format_option> stands for the option that completes the function. For example:

```
diagram_frame_height_reset()
table_print_column_reset()
```

All _reset functions have a return type of void. These functions are not listed in Table 2.

Two other _reset functions exist that reset all values to their defaults for diagrams and tables respectively:

```
diagram_reset_all()
table_reset_all()
```

These functions are listed in Table 2.

_override and _override_clear Functions

The _set functions cannot override values that are set with named settings. QRL provides a set of functions that can override some named setting values. These are the _override functions. These functions are listed in Table 2.

Just as QRL provides the _reset functions to undo the effects of the _set functions, QRL also provides a set of _override_clear functions to undo the effects of the _override functions. All _override functions have a corresponding _override_clear function. The _override_clear functions have the form:

```
void diagram_<format_option>_override_clear()
void table_<format_option>_override_clear()
```

where <format_option> stands for the option text that completes the function. For example:

```
diagram_frame_height_override_clear()
table_print_column_indices_override_clear()
```

All _override_clear functions have a return type of void. These functions are *not listed* in Table 2.

Two other _override_clear functions exist that reset all values to their named setting values for diagrams and tables respectively:

```
diagram_override_clear_all()
table_override_clear_all()
```

These functions are listed in Table 2.

QRL Built-in Functions

Table 2 lists and briefly describes all built-in functions supported by QRL.

Table 2: QRL Built-in Functions

Function	Function Type	Return Type	Description
app_type_print_string (<PDM_type> <object>)	OMS Interface	string	Returns a printable representation of the <object>'s application type.
append_file(string <the_file>, string <string_to_add>)	File	void	Appends <string_to_add> to <the_file>.
browse(< query_obj>)	OMS Interface	void	Invokes the Repository Browser and displays objects specified in a query (<query_obj>).
character_format_set (string <character_format>)	Formatting & printing text	string	Changes character format of text; returns current character format.
copy_file(string <from>, string <to>)	File	int	Copies file <from> to file <to>. Returns -1 upon failure.
current_projdir()	System	string	Returns the current project directory (may be different from the value of the *projdir* ToolInfo variable if that variable has been overridden).
current_system()	System	string	Returns the current system (may be different from the value of the *system* ToolInfo variable if that variable has been overridden).
delete_file(string <file_name>)	System	void	Deletes the <file_name> file.

Table 2: QRL Built-in Functions (Continued)

Function	Function Type	Return Type	Description
diagram_caption_alignment_set (alignment_enum <value>)	Diagram options	alignment _enum	Sets the horizontal adjustment of the caption.
diagram_caption_orientation_set (caption_orientation_enum <value>)	Diagram options	caption_ orienta- tion_ enum	Sets the orientation of the diagram caption. Permitted values are Portrait, Landscape, and AsDiagram.
diagram_caption_paragraph_ format_set(string <value>)	Diagram options	string	Sets the paragraph format of the diagram caption.
diagram_caption_placement_set (placement_enum <value>)	Diagram options	placement _enum	Sets the vertical placement of the caption. (RTF is restricted to Below).
diagram_display_mark_character_ format_set (string <value>)	Diagram options	string	Sets the character format of display marks.
diagram_display_mark_no_print_set (set <value>)	Diagram options	set	Specifies that a user-defined set of display marks is not to be displayed on the printed diagram.
diagram_display_mark_no_print_ override (set <value>)	Diagram options	set	Overrides the suppression of display marks.
diagram_frame_height_override (float <value>)	Diagram options	float	Overrides the frame height of a diagram set in a named setting.
diagram_frame_height_set (float <value>)	Diagram options	float	Sets the height of the diagram frame in current units.
diagram_frame_width_override (float <value>)	Diagram options	float	Overrides the frame width of a diagram set in a named setting.
diagram_frame_width_set (float <value>)	Diagram options	float	Sets the width of the diagram frame in current units.
diagram_label_character_format_set (string <value>)	Diagram options	string	Sets the character format of the diagram label.

Table 2: QRL Built-in Functions (Continued)

Function	Function Type	Return Type	Description
diagram_mark_override(string mark_label)	Diagram options	string	Overrides the diagram mark in a named setting.
diagram_mark_set (string mark_label)	Diagram options	string	Sets the diagram as a mark, that is, as the destination of an HTML hyperlink.
diagram_orientation_scale_override (orientation_size_enum <value>)	Diagram options	orienta-tion_scale _enum	Overrides the page orientation in a named setting.
diagram_orientation_scale_set (orientation_scale_enum <value>)	Diagram options	orienta-tion_scale _enum	Sets the orientation and scaling mechanism of the diagram on the page. Permitted values are PortraitScale, LandscapeScale, PortraitFit, LandscapeFit, and BestFit.
diagram_override_clear_all()	Diagram options	void	Resets all values to their named setting values for diagrams.
diagram_position_from_left_ override (float <value>)	Diagram options	float	Overrides the diagram left offset set in a named setting.
diagram_position_from_left_set (float <value>)	Diagram options	float	Sets the position, in current units, of the diagram from the left of the margin.
diagram_position_from_top_ override (float <value>)	Diagram options	float	Overrides the diagram top offset set in a named setting.
diagram_position_from_top_set (float <value>)	Diagram options	float	Sets the position, in current units, of the diagram from the top margin.
diagram_position_override (position_enum <value>)	Diagram options	position_ enum	Overrides the position of a diagram set in a named setting.
diagram_position_set (position_enum <value>)	Diagram options	position_ enum	Sets the position of the diagram relative to the margins.

Table 2: QRL Built-in Functions (Continued)

Function	Function Type	Return Type	Description
diagram_reset_all()	Diagram options	void	Resets all diagram values to their defaults.
diagram_show_caption_override (boolean <value>)	Diagram options	boolean	Overrides the diagram_show_ caption_set setting.
diagram_show_caption_set (boolean <value>)	Diagram options	boolean	Specifies whether diagram caption is shown.
diagram_scale_percent_override (float <value>)	Diagram options	float	Overrides the diagram size in a named setting.
diagram_size_percent_set (float <value>)	Diagram options	float	Sets diagram size in percent of diagram shown in editor at 100%. Not applicable if diagram_scale_orientation_set setting is PortraitFit, LandscapeFit, or BestFit.
directory_files (string <directory>, string <pattern>)	File	list	Returns a list of files in <directory> matching <pattern>. If <pattern> is NULL or "", "*" is used as the pattern.
document(string [file_name],string[title])	Printing documents	void	For HTML document generation. When called before any QRL output function is called, specifies the document's first output file name and title. When called after any QRL output function is called, begins a new document, using supplied arguments. If the file_name does not include a relative or absolute path, the file is placed in the same folder as the previous output file.

Table 2: QRL Built-in Functions (Continued)

Function	Function Type	Return Type	Description
enum_first (string <enum_type_name>)	Enumeration type	<enum_ type>	Returns the first element of the enumeration type.
enum_last (string <enum_type_name>)	Enumeration type	<enum_ type>	Returns the last element of the enumeration type.
enum_next (<enum_type> <element>)	Enumeration type	<enum_ type>	Returns the next <element> of the enumeration type or NULL if the <element> is the last of this enumeration type.
enum_prev (<enum_type> <element>)	Enumeration type	<enum_ type>	Returns the previous <element> of the enumeration type or NULL if the <element> is the first of this enumeration type.
environment_variable (string <the_variable>)	System	string	Returns the value of <the_variable>.
file_exists(string <the_file>)	File	boolean	True if <the_file> exists, otherwise False.
file_name_expand(string <the_file>)	System	string	Returns the fully exanded filename by searching through the *<product>_stp_file_path* variable.
file_part(string <path>)	File	string	Returns the filename part of <path>. If <path> has no file_part, "" is returned.
file_print(file <file_object>, string <caption>)	Printing tables & diagrams	void	Prints a table or diagram where <file_object> must be a table or diagram file.
file_print(file <file_object>, string <caption>, string <named_setting>)	Printing tables & diagrams	void	Prints a table or diagram where <file_object> must be a table or diagram file. Uses the specified <named_setting> to override the script's option values.

Table 2: QRL Built-in Functions (Continued)

Function	Function Type	Return Type	Description
file_print_named_settings_use (boolean <value>)	Printing tables & diagrams	void	Turns on or off the ability of a script to use the named setting associated with the table or diagram being printed.
file_print_units_set (units_enum <units>)	Printing tables & diagrams	units_ enum	Sets the units as either inches (Inches) or centimeters (Cm).
file_print_units_set (float <points_per_unit>)	Printing tables & diagrams	float	Sets the number of points per unit when a unit other than Inches or Cm is desired.
find_by_query (string <oms_query>)	OMS Interface	<PDM_ type>	Returns the first object matching the query or NULL if no match.
for_each_in_select (string <oms_query>, <PDM_type> <query_obj>)	OMS Interface	<PDM_ type>	Retrieves repository objects specified by <oms_query>.
format()	Printing tables & diagrams	string	Returns the name of the format file used to format the script's output.
format(string <format_file>)	Printing tables & diagrams	string	Specifies the format file to be used to format the script's output. Returns the name of the format file. <format_file> may be a string or a string variable.
getwd()	System	string	Returns the absolute pathname of the current working directory.
getpid()	System	int	Returns the process id of the calling process.
graph_clear(graph <the_graph>)	Graph	none	Clears the_graph.

Table 2: QRL Built-in Functions (Continued)

Function	Function Type	Return Type	Description
graph_copy(graph <the_graph>)	Graph	graph	Returns a copy of <the_graph>.
graph_count(graph <the_graph>)	Graph	int	Returns the number of nodes in <the_graph>.
graph_create()	Graph	graph	Creates an empty graph.
graph_create(set <the_set>)	Graph	graph	Creates a graph from the set of strings (that is, it creates a graph of *N* nodes, whereby each node is given a label from the set of strings).
graph_create(list <the_list>)	Graph	graph	Creates a graph from the list of strings (that is, it creates a graph of *N* nodes whereby each node is given a label from the list of strings).
graph_add_node (graph <the_graph>, string <label>)	Graph	none	Adds a node with label to <the_graph> (if it does not already exist).
graph_is_node(graph <the_graph>, string <label>)	Graph	boolean	True if a node with a label exists in <the_graph>, otherwise False.
graph_all_nodes (graph <the_graph>)	Graph	list	Returns a list of strings of the labels of all the nodes in <the_graph>.
graph_add_arc(graph <the_graph>, string <from_label>, string <to_label>)	Graph	none	Adds an edge from node <from_label> to node <to_label>. If either node does not exist, then they are implicitly added.
graph_is_arc(graph <the_graph>, string <from_label>, string <to_label>)	Graph	boolean	True if an edge from node <from_label> to node <to_label> exists, otherwise False.

Table 2: QRL Built-in Functions (Continued)

Function	Function Type	Return Type	Description
graph_is_indirect_arc (graph <the_graph>, string <from_label>, string <to_label>)	Graph	boolean	True if there is a direct / indirect edge from node <from_label> to node <to_label> exists, otherwise False.
graph_node_predecessors (graph <the_graph>, string <to_label>)	Graph	list	Returns a list of strings of the labels of all the nodes that have an edge going to node <to_label>.
graph_node_successors (graph <the_graph>, string <from_label>)	Graph	list	Returns a list of strings of the labels of all the nodes that have an edge coming from node <from_label>.
graph_node_indirect_predecessors (graph <the_graph>, string <to_label>)	Graph	list	Returns a list of strings of the labels of all the nodes that have a direct / indirect edge going to node <to_label>.
graph_node_indirect_successors (graph <the_graph>, string <from_label>)	Graph	list	Returns a list of strings of the labels of all the nodes that have a direct / indirect edge coming from node <from_label>.
graph_node_rank (graph <the_graph>, string <label>)	Graph	int	Returns the rank of node labelled <label>.
graph_topological_sort (graph <the_graph>)	Graph	list	Returns a list of strings of the labels of all the nodes in <the_graph> in topological order (that is, by rank).
graph_is_acyclic (graph <the_graph>)	Graph	boolean	True if the graph has no cycles, otherwise False.
home_directory(void)	File	string	
hostname()	System	string	Returns hostname.
id_list_create(string <oms_query>, string <id_list_name>)	OMS Interface	void	Creates a list of the IDs of currently selected objects.

Table 2: QRL Built-in Functions (Continued)

Function	Function Type	Return Type	Description
id_list_free(string <id_list_name>)	OMS Interface	void	Deletes list that was created to contain object IDs.
list_append(list <the_list>, <any type> <item>)	List type	int	Appends an item to the end of <the_list>, and returns <the_list> length. If <item> is of the wrong type, an error is flagged.
list_clear(list <the_list>)	List type	void	Clears <the_list> and frees the space.
list_concatenate(list <list1>, list< list2>)	List type	int	Concatenates a deep copy of <list2> to <list1> and stores the result in <list1>. Returns <list1> length. If the lists are not of the same type, an error is flagged.
list_copy(list <the_list>)	List type	list	Returns a copy of <the_list>.
list_count(list <the_list>)	List type	int	Returns list length.
list_create (string <member_type>, int <initial_size>)	List type	list	Creates a list whose items are of <member_type>. If "all" is used as the value of <member_type>, the list is heterogeneous; if <initial_size> > 0, the list is created to be of that size, and padded with NULLs.
list_delete(list <the_list>, int <index>)	List type	int	Deletes the item at position <index> in <the_list>. Items <index> + 1 through last become items <index> through last - 1. Returns list length. If <index> > last or < 0, an error is flagged; unused space is freed.

Table 2: QRL Built-in Functions (Continued)

Function	Function Type	Return Type	Description
list_find(list <the_list>, int <start_index>,<item>)	List type	int	Returns index of <item> in <the_list>. The search starts at <start_index>; returns count if not found. An error is flagged if the item type is not the same as the type stored in <the_list>, or if <start_index> is out of range.
list_get(list <the_list>, int <index>)	List type	all	Returns the value of the item at position <index> in <the_list>. If <index> is out of range, an error is flagged.
list_get_type(list <the_list>, int <index>)	List type	string	Returns the type of the item at position <index> in <the_list>. If <index> is out of range, an error is flagged.
list_insert(list <the_list>, int <index>, <all item>)	List type	int	Inserts an item before index. Items index through last become items (index + 1) through (last + 1). Returns list length. If index > last or < 0, an error is flagged. If item is of the wrong type, an error is flagged.
list_of_diagrams_print (string render_format)	List type	void	For HTML document generation. Prints the List of Diagrams. Each entry corresponds to a diagram's caption. Text is rendered in the paragraph format supplied as the argument.

Table 2: QRL Built-in Functions (Continued)

Function	Function Type	Return Type	Description
list_of_tables_print(string render_format)	List type	void	For HTML document generation. Prints the List of Tables. Each entry corresponds to a table's caption. Text is rendered in the paragraph format supplied as the argument.
list_select(string <oms_query>)	List type, OMS interface	list	Executes <oms_query> and returns a list containing the results. The list type corresponds to the query.
list_set(list <the_list>, int <index>, <all value>)	List type	void	Sets the item at position <index> in <the_list> and frees the previous item at <index>. If <index> is out of range or the value is of the wrong type, an error is flagged.
list_sort(list <the_list>)	List type	list	Sorts a homogeneous list, <the_list>, in alphabetical order. Returns NULL if the list elements are not all of the same type.
list_sort_all(list <the_list>, string <qrl_compare_fn>)	List type	list	Sorts a homogeneous list, <the_list>, using <qrl_compare_fn>. Returns NULL if the list elements are not all of the same type.
list_sort_by_field(list <the_list>, string <qrl_compare_fn>)	List type	list	Sorts a heterogeneous list, <the_list>, using <qrl_compare_fn>. Returns NULL if the list elements are not all of the same type.

Table 2: QRL Built-in Functions (Continued)

Function	Function Type	Return Type	Description
list_to_string(list <the_list>, string <separator>)	List type	string	Converts a list to a string with the <separator> string separating the string representation of each item in <the_list>.
lists_equal(<list1>, <list2>)	List type	boolean	Tests whether two lists have the same type and the same values in the same order.
main()		void	Starts every QRL script.
message(<all value>)	System	void	Outputs value to the message log or stdout.
mkdir(string <directory>)		int	Makes a directory named <directory>. Returns -1 upon failure.
page()	Formatting & printing text	string	Starts a new page, using the paragraph format specified for the previous paragraph. Returns the previous paragraph format.
page(string <paragraph_format>)	Formatting & printing text	string	Starts a new page using the specified paragraph format. Returns the previous paragraph format.
page_equals_document (target_enum a_target)	Printing documents	void	HTML Only. Specifies that each call to page is identical to a call to the document function with empty arguments.
paragraph()	Formatting & printing text	string	Starts a new paragraph using the paragraph format specified for the previous paragraph format. If none was specified, uses the default. Returns the previous paragraph format.

Table 2: QRL Built-in Functions (Continued)

Function	Function Type	Return Type	Description
paragraph (string <paragraph_format>)	Formatting & printing text	string	Changes paragraph format and starts a new paragraph. If the paragraph definition includes starting a new page, then this creates a page break. Returns old paragraph format.
paragraph_end()	Formatting & printing text	void	Ends the current paragraph. Necessary for generating nested HTML lists. For an example, see "HTML Example" on page 5-36
parent_directory (string <path>)	File	string	Returns the parent directory of <path>.
path_compose (list <components>)	File	string	Returns a path composed of components. Some normalization is done so that there is exactly one delimiter between components.
path_compose (string <component1>, string <component2>....)	File	string	Returns a path composed of components. Some normalization is done so that there is exactly one delimiter between components.
path_last_delimiter (string <path>)	File	int	Returns the position in <path> of the last path delimiter character. The length of the string <path> is returned if there are no delimiter characters.
path_normalize(string <path>)	File	string	Makes all delimiters in <path> look the same.

Table 2: QRL Built-in Functions (Continued)

Function	Function Type	Return Type	Description
path_part(string <path>)	File	string	Returns the path prefix of <path> including the final path delimiter. If <path> has no path part, "" is returned.
print(<all value>)	Formatting & printing text	void	Prints the value where <value> can be any legal expression. Does not add a hard return. This is the preferred function to use with paragraph formats that wrap (such as HTML, FrameMaker, and RTF paragraph formats, and ASCII paragraphs if LongLine-Handling is set to Wrap).
print_line()	Formatting & printing text	void	Starts a new line.
print_line(<all value>)	Formatting & printing text	void	Prints the value and starts a new line; where <value> can be any legal expression.
print_link(<all types> <value>, string <mark_label>)	Formatting & printing text	void	HTML only. Prints values in the identical manner as the print function. In addition, the value becomes the source of an HTML hyperlink whose destination is given by the string mark_label.

Table 2: QRL Built-in Functions (Continued)

Function	Function Type	Return Type	Description
print_mark(<all types> <value>, string <mark_label>)	Formatting & printing text	void	HTML only. Prints values in the identical manner as the print function. In addition, the value becomes a mark, that is, an anchor to serve as the destination of an HTML hyperlink. Each mark must have a unique mark_label.
print_raw(string <value>)	Formatting & printing text	void	For non-ASCII output (that is, FrameMaker, HTML, or Microsoft Word), does not add any formatting statements to the output document, and prints the value without escaping any character sequences meaningful to the target. (**Note:** all other print commands escape meaningful sequences.)
printf(string format,... list of expressions)	Formatting & printing text	void	Like C **printf** function. Translates values to characters. Converts, formats, and prints the values as substituted in arguments prefixed by %. QRL supports most of the same formatting characters and options as the C language.
put_environment_variable (string <the_variable>)	System	int	Adds <the_variable> and its value to the environment.
read_dir_access(string <the_dir>)	File	boolean	True if the directory can be read, otherwise False.
read_file(string <file_name>)	File	string	Returns the contents of <file_name>.

Table 2: QRL Built-in Functions (Continued)

Function	Function Type	Return Type	Description
read_file_access(string <the_file>)	File	boolean	True if the file can be read, otherwise False.
repos_add_maker(string <users>, {string <server>}, {string <password>})	Repository	int	Grants system creation privileges to one or more Sybase users. Automatically adds the specified users to the server. You do not need to call repos_add_user separately for each new user. Use the <password> parameter if the server is secure or partially secure and the user is a new user of the server. If you do not specify a password, users will be prompted to enter one.
repos_add_user(string <users>, {string <server>}, {string <password>})	Repository	int	Adds one or more Sybase users to the server, which allows them to connect to an StP system. If you do not specify a password, users will be prompted to enter one.
repos_can_list_current_users ({string <server>})	Repository	boolean	Returns True if the server can list current users. This function returns True for Sybase and False for Microsoft Jet.
repos_can_modify_users({string <server>})	Repository	boolean	Returns True if the server can modify user information. If a server returns False for this function, it will not support repos_add_user, repos_delete_user, and repos_change_password. This function is True for Sybase and False for Microsoft Jet.

Table 2: QRL Built-in Functions (Continued)

Function	Function Type	Return Type	Description
repos_change_password(string <users>, {string <server>}, {string <password>})	Repository	int	Changes one or more Sybase server users' passwords to the specified password. Users who do not have administrative privileges can only use repos_change_password to change their own passwords.
repos_delete_maker(string <users>, {string <server>})	Repository	int	Revokes system creation privileges from one or more Sybase users.
repos_delete_user(string <users>, {string <server>})	Repository	int	Removes users from the server. Users can no longer connect to StP systems on a server from which they have been removed. repos_delete_user also removes the users from any systems they formerly had access to. You do not need to call sys_delete_user separately.
repos_has_dynamic_space({string <server>})	Repository	boolean	Returns True if databases on the server will expand automatically as necessary; otherwise, it returns False. If this function returns True for a server, that system will not support the repos_show_space function. All systems on that server will not support sys_expand_rep or sys_show_space. This function returns False for Sybase and True for Microsoft Jet.
repos_list_reps({string <server>})	Repository	int	Lists all repositories under the server.

Table 2: QRL Built-in Functions (Continued)

Function	Function Type	Return Type	Description
repos_list_users({string <server>})	Repository	int	Displays the names of all valid server users. These users do not need to be currently using the server.
repos_list_current_users({string <server>})	Repository	int	Displays the names of all valid users currently using the server.
repos_maint({string <server>})	Repository	int	Performs routine server maintenance, including deleting old temporary tables, dumping the server's transaction log, and updating the statistics used by the Adaptive Server or SQL Server indexes and optimizer.
repos_show_space({string <server>}, {optional_bool_enum <all>})	Repository	int	Shows the available space, in megabytes, in the Sybase repository. The <all> parameter shows space for all of the devices in the Sybase server.
repos_supports_makers({string <server>})	Repository	boolean	Returns True if the server supports repos_add_maker and repos_delete_maker. This function is True for Sybase and False for Microsoft Jet.
root_part(string <path>)	File	string	Returns the root part of <path>. If <path> has no root part, NULL is returned.
selection_count (string <oms_query>)	OMS interface	int	Returns the number of objects selected by <oms_query>.

Table 2: QRL Built-in Functions (Continued)

Function	Function Type	Return Type	Description
set_add(set <the_set>, <all element>)	Set type	int	If <element> is not a member of <the_set>, adds <element> to the set and returns <the_set> cardinality (size). If <element> is of the wrong type, an error is flagged.
set_clear(set <the_set>)	Set type	void	Empties <the_set>.
set_copy(set <the_set>)	Set type	set	Returns a deep copy of <the_set>.
set_count(set <the_set>)	Set type	int	Returns the cardinality of <the_set>.
set_create(string <member_type>)	Set type	set	Creates an empty set whose elements are of type <member_type>. If "all" is used as the value of <member_type>, the set is heterogeneous.
set_delete(set <the_set>, <all element>)	Set type	int	If <element> is a member of <the_set>, delete <element>. Returns <the_set> cardinality (size).
set_difference(set <set1>, set <set2>)	Set type	int	Stores the difference of <set1> and <set2> into <set1>; returns the cardinality of <set1>. If sets are not of the same type, an error is flagged.
set_get_element(set <the_set>, int <nth>)	Set type	all	Returns the value of the <nth> element in <the_set>. If <nth> < 0 or >= to <the_set> cardinality, an error is flagged.

Table 2: QRL Built-in Functions (Continued)

Function	Function Type	Return Type	Description
set_get_type(set <the_set>, int <nth>)	Set type	string	Returns the type of the <nth> element in <the_set>. If <nth> < 0 or >= <the_set> cardinality, an error is flagged.
set_intersection(set <set1>, set <set2>)	Set type	int	Stores the intersection of <set1> and <set2> into <set1>; returns the cardinality of <set1>. If sets are not of the same type, an error is flagged.
set_is_member(set <the_set>, <all element>)	Set type	boolean	Returns True if <element> is in <the_set>, False otherwise.
set_is_proper_subset(set <set1>, set <set2>)	Set type	boolean	Returns True if <set2> is a proper subset of <set1>.
set_is_subset(set <set1>, set <set2>)	Set type	boolean	Returns True if <set2> is a subset of <set1>.
set_select(string <oms_query>)	Set type, OMS interface	set	Executes <oms_query> and returns a set containing the result. The returned set is of the type corresponding to the query.
set_union(set <set1>, set <set2>)	Set type	int	Stores the union of <set1> and <set2> into <set1>; returns the cardinality of <set1>. If sets are not of the same type, an error is flagged.
sets_equal(set <set1>, set <set2>)	Set type	boolean	Tests whether two sets have the same type and elements. Returns True or False.
skip_line(int <lines>)	Formatting & printing text	void	Skips number of <lines> specified in argument.

Table 2: QRL Built-in Functions (Continued)

Function	Function Type	Return Type	Description
string_converts(string <the_string>, string <target_type>)	String manipulation	boolean	Tests whether the string legally converts to a value of <target_type> (<target_type> must be one of the primitive types).
string_escape(string <the_string>, string <badchars>)	String manipulation	string	Escapes specified <badchars> in <the_string> and converts newlines to \ \n. Returns string with <badchars> escaped.
string_extract(string <the_string>, int <start>, int <length>)	String manipulation	string	Extracts the substring starting at <start> of <length>.
string_find(string <the_string>, int <start>, string <the_substring>)	String manipulation	int	Returns the index into <the_string> where <the_substring> occurs. Starts the search at <start>. Returns the length of <the_string> if <the_substring> is not found.
string_length(string <the_string>)	String manipulation	int	Returns the length of <the_string>.
string_search_and_replace (string <the_string>, string <search_for>, string <replace_by>)	String manipulation	string	Returns a string with all occurrences of <search_for> in <the_string> replaced by the <replace_by> string.
string_strip(string <the_string>, string <option>, string <stripchars>)	String manipulation	string	Strips specified <stripchars> according to the value specified for <option>: L (leading), T (trailing), or B (both). Returns string with <stripchars> removed.

Table 2: QRL Built-in Functions (Continued)

Function	Function Type	Return Type	Description
string_to_list(string the_string, string <separator>s)	List type	list	Converts a list to a string with the <separator> string separating the string representation of each item in <the_list>.
string_to_time(string <time>, string <fmt>)	System	int	Returns integer representation of <time>; <fmt> parameter works the same way C-library <strftime fmt> parameter works. If <fmt> is NULL, a default is used.
string_translate(string <the_string>, string <input_table>, string <output_table>)	String manipula-tion	string	Translates <the_string> by substituting the value specified in <output_table> for corresponding value specified in <input_table>. Returns translated string.
sys_add_user({string <projdir>}, {string <system>}, {string <writers>}, {string <readers>})	System repository	int	Adds Sybase users to a system. In a secure system, the executor must have at least owner permissions.
sys_can_list_current_users({string <projdir>}, {string <system>})	System repository	boolean	Returns True if the system can list current users, otherwise False. If False, the server that the system is on will not support repos_list_current_users. This function returns True for Sybase systems and False for Microsoft Jet systems.

Table 2: QRL Built-in Functions (Continued)

Function	Function Type	Return Type	Description
sys_can_modify_users({string <projdir>}, {string <system>})	System repository	boolean	Returns True if the system can modify user information. If a system returns False for this function, it will not support sys_add_user, sys_delete_user, sys_change_user, and sys_change_repowner. The server that the system is on will not support repos_add_user, repos_delete_user, and repos_change_password. This function returns True for Sybase systems and False for Microsoft Jet systems.
sys_change_repowner({string <projdir>}, {string <system>}, string <user>, {string <password>})	System repository	int	Changes the owner of the system's Sybase repository and adds the specified new user to the server. You do not need to call repos_add_user separately. Use the <password> parameter to set the new user's password in the server. This function is not valid for Microsoft Jet.
sys_change_user ({string <projdir>}, {string <system>}, {string <writers>}, {string <readers>})	System repository	int	Changes the security level between reader and writer Sybase users. Using the <writers> parameter grants read and write privileges. Using the <readers> parameter grants read-only privileges. The executor must be at least have owner permissions in a secure system.

Table 2: QRL Built-in Functions (Continued)

Function	Function Type	Return Type	Description
sys_clean_rep({string <projdir>}, {string <system>}, {optional_bool_enum <force>})	System repository	int	Deletes all data from the system's repository, but does not destroy the repository. It does not affect files in the system's directory. In a secure system, the executor must be at least an owner. Users must not be running any editors (GDE, GTE, OAE) while this command is running.

Table 2: QRL Built-in Functions (Continued)

Function	Function Type	Return Type	Description
sys_create({string <projdir>}, {string <system>}, {string <repositoryType>}) and sys_create({string <projdir>}, {string <system>}, {string <repositoryType>}, {int <size>}, {string <server>}, {string <filemode>}, {string<repname>)	System repository	int	Creates a new system, including its *.repinfo* file and directories for its various editors' flat files. The executor, who must be a system administrator (or maker; see repos_add_maker) in a secure environment, becomes the owner of the new system and the database owner of the system repository's underlying database. When a new system is created, locking is automatically enabled, and the system owner is the lock administrator. To disable locking or to change the lock administrator, use the **Locks** submenu available from the **Tools** menu on the Desktop. For Sybase, the default setting for the <size> parameter is 6. By default, sys_create provides an additional 2 MB for the system's transaction log. You can reset the transaction log size with the *syscreate_log_size* ToolInfo variable. The default for <filemode> is c (change).
sys_delete_user({string <projdir>}, {string <system>}, string <users>)	System repository	int	Denies Sybase users access to the specified system. In a secure system, the executor must at least have owner permissions.

Table 2: QRL Built-in Functions (Continued)

Function	Function Type	Return Type	Description
sys_destroy({string <projdir>}, {string <system>}, {optional_bool_enum <force>})	System repository	int	Destroys a system's repository and all files in the system directory. The system cannot be regenerated after being destroyed with this command. In a secure system, the executor must at least have owner permissions.
sys_destroy_rep({string <projdir>}, {string <system>}, boolean <force>)	System repository	int	Destroys a system's repository, but does not remove the files from the system directory. The repository can be regenerated from the files after being destroyed by this command. In a secure sytem, the executor must at least have owner permissions.

Table 2: QRL Built-in Functions (Continued)

Function	Function Type	Return Type	Description
sys_dump_rep({string <projdir>}, {string <system>}, string <dumpdir>, {optional_bool_enum <dumpuserinfo>})	System repository	int	For Sybase only. Dumps a system's repository to special dump files that can be subsequently reloaded with the sys_load_rep function or from the Desktop **Repository** menu (from the **Repository** menu, choose **Maintain Systems** > **Load Current System Repository from Files**). Overwrites any existing files in the dump directory. A repository can be dumped while users are accessing it. The dump reflects the state of the data at the time the dump began. Dump a repository to provide a backup that can be reloaded in the event of media failure, or to copy a repository to another system or another machine.
sys_duplicate({string <oldProjdir>}, string <oldSystem>, {string <newProjdir>}, string <newSystem>)	System repository	int	Creates a duplicate copy of an existing system with a new name.
sys_expand_rep({string <projdir>}, {string <system>}, {int <size>})	System repository	int	For Sybase only. Expands the system repository by allocating additional space for its use on the device specified by the *syscreate_device* ToolInfo variable. The default expansion is 2 MB, and the default device is "default." In a secure system, the executor must be at least the system administrator or have owner permissions.

Table 2: QRL Built-in Functions (Continued)

Function	Function Type	Return Type	Description
sys_get_rep_type({string <projdir>}, {string <system>})	System repository	string	Displays the repository type for a given system, as specified in the *.repinfo* file. You can use the output of this function as a parameter for the sys_create function. Possible values are "Sybase" and "MS Jet." If you make an error, it returns an empty string.
sys_has_dynamic_space({string <projdir>}, {string <system>})	System repository	boolean	Returns True if the system database can expand automatically as necessary. If the system cannot expand automatically, this function returns False. Systems for which this function returns True do not support the sys_expand_rep or sys_show_space function. The server that the system is on will not support repos_show_space. This function returns False for Sybase systems and True for Microsoft Jet systems.
sys_has_rep({string <projdir>}, {string <system>})	System repository	boolean	Verifies whether a system has a repository.
sys_list_current_users({string <projdir>}, {string <system>})	System repository	int	For Sybase only. Lists all users currently accessing the system.
sys_list_users({string <projdir>}, {string <system>})	System repository	int	Lists all users who may gain access to the named system.

Table 2: QRL Built-in Functions (Continued)

Function	Function Type	Return Type	Description
sys_load_rep({string <projdir>}, {string <system>}, {string <dumpdir>}, {optional_bool_enum <force>}, {optional_bool_enum <fast>}, {optional_bool_enum <loaduserinfo>})	System repository	int	For Sybase only. Loads a system's repository from a set of dump files created by the sys_dump_rep function or from the Desktop **Repository** menu (from the **Repository** menu, choose **Maintain Systems** > **Dump Current System Repository to Files**). Dump files are different from the system's flat files. Dump files are binary representations of the information in the repository. They contain information such as how StP objects are connected and what their attributes are. The system's flat files are human-readable diagram, table, and annotation files. All dump file information is also in the flat files, but the converse is not true; the flat files also contain information such as diagram layouts, which is not stored in the repository. A repository cannot be loaded while users are accessing the system. The target system does not have to be the system from which the dump files were dumped. In a secure system, the executor must be at least an owner.

Table 2: QRL Built-in Functions (Continued)

Function	Function Type	Return Type	Description
sys_recover_rep({string <projdir>}, {string <system>}, {optional_bool_enum <force>}, {optional_bool_enum <fast>}) and sys_recover_rep({string <projdir>}, {string <system>}, {string <repositoryType>}, {int <size>}, {string <server>}, {int <filemode>}, {string <repname>}, {optional_bool_enum <force>}, {optional_bool_enum <fast>})	System repository	int	Regenerates a system repository from the system files. During regeneration, the system is locked, preventing all other system administration functions from being run concurrently. This is accomplished by placing a file, *.sysbusy*, at the system level, which is removed once the system has been rebuilt. sys_recover_rep calls sys_update, and all the same conditions hold with respect to potential writability of files. The executor must be a system administrator or system owner and must have write permissions on the files.
sys_show_space({string <projdir>}, {string <system>})	System repository	int	For Sybase, shows the size of the current system, available megabytes, and percent full. Display contains separate lines for the repository and transaction log.
sys_supports_rep_dump({string <projdir>}, {string <system>})	System repository	boolean	Returns True if the system supports sys_dump_rep and sys_load_rep. This function returns True for Sybase and False for Microsoft Jet systems.

Table 2: QRL Built-in Functions (Continued)

Function	Function Type	Return Type	Description
sys_trunc_hist({string <projdir>}, {string <system>}, {optional_bool_enum <keepusedonly>})	System repository	int	Whenever a user creates, updates, saves, or views a file, a file history object is created in the repository to note the action. Over time, these objects accumulate. sys_trunc_hist prunes all but the most recent file history object for each file. If you set <keepusedonly> to True, it prunes (per file) all but the most recent file history object for each user. In a secure system, the executor must be at least an owner.
sys_update [{string <projdir>}, {string <system>}, {optional_bool_enum <besteffort>}, {optional_bool_enum <force>}, {string <filenames>}, {string <filetypes>})	System repository	int	Adds new files to the repository and updates out-of-date files. sys_update updates the repository immediately, then updates the system files the next time they are loaded and saved in an editor. sys_update updates all system files by processing any pending renames. sys_update calls the **aupdate**, **tupdate**, and **dupdate** commands (which are described in *StP Administration*). The executor must have write access to all files to be processed. If an out-of-date file is under version control and is either checked out by another user or does not have proper file permissions, the command fails. The sys_update function replaces the sysbuildrep function.

Table 2: QRL Built-in Functions (Continued)

Function	Function Type	Return Type	Description
system (string <operating_system_cmd>)	System	void	Executes operating system command.
tab(int <tabs>)	Formatting & printing text	void	"Tabs" over the number of tab stops specified in argument.
table_broken_ruling_set (string <value>)	Table options	string	Sets the "broken" ruling to a different ruling type.
table_caption_alignment_set (alignment_enum <value>)	Table options	alignment _enum	Sets the horizontal placement of the table caption.
table_caption_paragraph_format_set (string <value>)	Table options	string	Sets the paragraph format of the table caption.
table_caption_placement_set (placement_enum <value>)	Table options	placement _enum	Sets the vertical placement of the table caption.
table_cell_bold_paragraph_format_ set(string <value>)	Table options	string	Sets the paragraph format for bold text in table cells.
table_cell_paragraph_format_set (string <value>)	Table options	string	Sets the paragraph format for text in table cells.
table_column_range_override(string <value>)	Table options	string	Overrides the table body columns setting in a named setting.
table_column_range_set (string <value>)	Table options	string	Sets which table columns to print. Specified as a comma-separated list of intervals that are ordered and do not over-lap, for example "1-3, 5, 8-*"
table_double_ruling_set (string <value>)	Table options	string	Sets the "double" ruling to a different ruling type.
table_header_row_range_override (string <value>)	Table options	string	Overrides the table header row setting in a named setting.

Table 2: QRL Built-in Functions (Continued)

Function	Function Type	Return Type	Description
table_header_row_range_set (string <value>)	Table options	string	Sets which rows are repeated at the top of every page if a table spans more than one page. Specified in same way as table_column_range_set.
table_height_override (float <value>)	Table options	float	Overrides the height of the table in in a named setting.
table_height_set (float <value>)	Table options	float	Sets the height of the table in current units. Default is 504 pts (7 inches).
table_mark_override(string mark_label)	Table options	string	Overrides the table as a mark in a named setting.
table_mark_set(string mark_label)	Table options	string	Sets the table as a mark, that is, as the destination of an HTML hyperlink.
table_maximum_vertical_span_ override (int <value>)	Table options	int	Overrides the table maximum vertical span setting in a named setting.
table_maximum_vertical_span_set (int <value>)	Table options	int	Sets the maximum number of rows a table cell can span.
table_medium_ruling_set (string <value>)	Table options	string	Sets the “medium” ruling to a different ruling type.
table_of_contents_entry (string entry_format, string render_format)	Formatting & printing text	void	For HTML document generation. Specifies the paragraph format to be included and how its text is rendered in the Table of Contents.
table_of_contents_print()	Formatting & printing text	void	For HTML document generation. Prints the Table of Contents.

Table 2: QRL Built-in Functions (Continued)

Function	Function Type	Return Type	Description
table_orientation_override (table_orientation_enum <value>)	Table options	table_orientation_enum	Overrides the table orientation setting in a named setting.
table_orientation_set (table_orientation_enum <value>)	Table options	table_orientation_enum	MIF or RTF only. Sets whether table is created in a portrait or landscape format. Default is PortraitTable.
table_override_clear_all()	Table options	void	Resets all values to their named setting values for tables.
table_print_column_indices_override (boolean <value>)	Table options	boolean	Overrides the table show column numbers setting in a named setting.
table_print_column_indices_set (boolean <value>)	Table options	boolean	Sets whether columns of the table are numbered in the output file.
table_print_row_indices_override (boolean <value>)	Table options	boolean	Overrides the table show row numbers setting in a named setting.
table_print_row_indices_set (boolean <value>)	Table options	boolean	Sets whether rows of the table are numbered in the output file.
table_reset_all()	Table options	void	Resets all table values to their defaults.
table_row_range_override (string <value>)	Table options	string	Overrides the table body rows setting in a named setting.
table_row_range_set(string <value>)	Table options	string	Sets which rows are printed in the output. Specified in same way as table_column_range_set.

Table 2: QRL Built-in Functions (Continued)

Function	Function Type	Return Type	Description
table_shade1_color_set (int <value>)	Table options	int	Sets color for table shadings. See the target publishing tool documentation for the meaning of these integer values.
table_shade2_color_set (int <value>)			
table_shade3_color_set (int <value>)			
table_shade1_pattern_set (int <value>)	Table options	int	Sets pattern for table shadings. See the target publishing tool documentation for the meaning of these integer values.
table_shade2_pattern_set (int <value>)			
table_shade3_pattern_set (int <value>)			
table_show_caption_override (boolean <value>)	Table options	boolean	Overrides the table caption show setting in a named setting.
table_show_caption_set (boolean <value>)	Table options	boolean	If True, table caption is shown.
table_thick_ruling_set (string <value>)	Table options	string	Sets the “thick” ruling to a different ruling type.
table_thin_ruling_set (string <value>)	Table options	string	Sets the “thin” ruling to a different ruling type.
table_width_set(float <value>)	Table options	float	Sets the width of the table in current units.
target()	Printing tables & diagrams	target_ enum	Returns target specified for script output.
temp_file(string <base>, string <type>)	System	string	Generates a unique filename. <base> is the program requesting the unique name; <type> is the type of file, in case the program needs to generate unique names of different types.

Table 2: QRL Built-in Functions (Continued)

Function	Function Type	Return Type	Description
time_now()	System	int	Returns the number of seconds since 1970 in Greenwich Mean Time.
time_to_string(int <time>, string <fmt>)	System	string	Returns string representation of <time>; <fmt> parameter works the same way C-library <strptime fmt> parameter works. If <fmt> is NULL, a default is used.
to_boolean(<int or string> value)	Type conversion	boolean	Converts the int or string <value> to a boolean <value>.
to_enum(string <value>, string <enum_type_name>)	Enumera-tion type	<enum_ type>	Returns an element named <value> of the type <enum_type_name>. Generates an error if the value does not correspond to an element of type <enum_type_name>.
to_enum(int <value>, string <enum_type_name>)	Enumera-tion type	<enum_ type>	Returns an element of type <enum_type_name> whose position is <value>. Generates an error if <value> is out of range for <enum_type_name>.
to_float(<int or string> value)	Type conversion	float	Converts the int or string <value> to a boolean <value>.
to_int(<enum_type> <element>)	Enumera-tion type	int	Returns the position of element (as in C, elements are numbered starting with 0).
to_int(<float, string, or boolean> value)	Type conversion	int	Converts the float, string, or boolean <value> to an integer.
to_lower(string <the_string>)	String manipula-tion	string	Makes <the_string> lowercase.

Table 2: QRL Built-in Functions (Continued)

Function	Function Type	Return Type	Description
to_oms_string(string <the_string>)	OMS interface	string	Escapes OMS special characters (that is, apostrophes) in <the_string>; applied to QRL strings that are to be used as string literals in OMS queries. Returns <the_string> with the special characters escaped.
to_string (any primitive type <value>)	Type conversion	string	Returns the name of the <element>.
to_upper(string <the_string>)	String manipula-tion	string	Makes <the_string> uppercase.
toolinfo_variable (string <toolinfo_var>)	System	string	Returns the value of <toolinfo_var>.
trace(string <symbol>)	QRL debugging	string	Enables tracing of functions and all trace_print() messages for <symbol>. <symbol>can be a specific function or one of the predefined symbols listed in Table 20 on page 3-93.
trace_file(string <output_file>)	QRL debugging	string	Directs trace output from the default (stderr) to <output_file>.
trace_print(string <symbol>, string <any_string>)	QRL debugging	string	Displays <any_string> if tracing is enabled for <symbol>.
untrace(string <symbol>)	QRL debugging	string	Disables the tracing of functions and the display of trace_print() messages for <symbol>. <symbol> can be a specific function or the predefined symbol “calls,” “builtins,” “returns,” or “all.”

Table 2: QRL Built-in Functions (Continued)

Function	Function Type	Return Type	Description
user()	System	string	Returns username.
write_file(string <path>, string <contents>)	File	int	Writes <contents> to a file (<path>) and returns the number of characters written to the file.
write_file_access(string <the_file>)	File	boolean	True if the file can be written, otherwise False.

Index

D

E

F

G

H

P

Q

R

S

T

U

V

W