R&S®QuickStep Test Executive Software

User Manual

1177622302

User Manual

Version 10

ROHDE&SCHWARZ

This manual applies to the following options:

- R&S QuickStep Test Executive Software (1528.9010.02)
- R&S QuickStep Test Sequencer Software (1528.9049.02)
- Development option for R&S QuickStep Test Executive Software (1528.9026.02)
- R&S QuickStep license dongle and key card (1528.9003.02)

The firmware of the instrument uses several valuable open source software packages. For information, see the "Open Source Acknowledgment" document, which is available for download from the customer web section on GLORIS, the global Rohde & Schwarz information system: https://extranet.rohde-schwarz.com.
Rohde & Schwarz would like to thank the open source community for their valuable contribution to embedded computing.

Mühldorfstr. 15, 81671 München, Germany
Phone: +49 89 41 29 - 0
Fax: +49 89 41 29 12 164
Email: info@rohde-schwarz.com
Internet: www.rohde-schwarz.com

1177.6223.02 | Version 10 | R&S®QuickStep

The following abbreviations are used throughout this manual: R&S® is abbreviated as R&S.

Contents

1 Welcome

Welcome to the R&S QuickStep Test Executive software! QuickStep provides a high-speed test sequencer in combination with a powerful graphical user interface for the parameterization and control of test execution. Test procedures are designed in a graphical editor as flow charts based on the provided or additionally developed test functions.

QuickStep lets you set up and run test plans – sequences of individual tests together with scheduling and execution information –, to build test procedures and to evaluate the test results. During test execution, QuickStep controls the test equipment.

QuickStep includes example test projects for typical test conditions and hardware setups. It offers facilities to adapt given test plans and their execution schedules and to develop new ones. Customer extensions can easily be integrated, e.g. for exploiting or developing special test algorithms.

If you only use the QuickStep OTA basic application (installed with QuickStep, QS-ATSCAL license required), most test executive features are hidden at the GUI but used in the background. For details, see the QuickStep ATSCAL OTA Testing user manual.

1.1 Key Features

General features:

- High performance:
  QuickStep causes a minimum processing overhead. The test execution speed is comparable to native C++/C# code. Parallel execution of code is supported.
- User-friendly handling:
  Configurations are done via graphical user interface (GUI) and intuitive handling (for example drag & drop). Standard tests need only a minimum configuration effort.
- High flexibility and wide application range:
  - Examples and reference test cases are available and ready to run.
    The test packages are optimized regarding time consumption.

  - QuickStep is not confined to a certain set of test instruments since standard communication interfaces are used for controlling the test equipment. 3rd-party instruments are easily integrated.
  - Customer-defined test setups and test conditions are supported.
  - QuickStep is appropriate for production tests (particularly due to high performance) as well as for test development purposes.
- Support for developing test functionality and easy integration of customer code

Test plan configuration:

- Static and dynamic parameter references
- Convenient set and sweep functions
- Search and filter functions
- Limit handling
- Control phases (for example for instrument initialization) around sequences of test steps
- Dynamic control statements (loops, if conditions, jumps)

Test results and execution protocol:

- Diagram for result plots
- Histogram view for statistical analysis
- Configurable result charts and live view results
- Configurable reports
- Test execution protocol viewer

System configuration management:

- Graphical representation of the test setup
- Intuitive building of system configurations with elements from a library
- Parameter and path mapping for multiple test setups

Development of new tests:

- Intuitive building of test procedures via flow charts of blocks from a library
- Control structures (conditions, "If", "Or") and dependencies
- GUI supported generation of source code templates for new test functions
- Powerful API to support standard functionality
- Microsoft Visual Studio® based test function development with C++ or C#

- Debugging support: Breakpoints allow to pause a test run on well-defined steps and block functions; single step execution mode is provided
- Re-use and extension of R&S Forum and MATLAB scripts
- Support of user-defined dialogs implemented with Windows Forms or WPF
- Standalone usage of block DLLs

1.2 Software Components, Product Licensing

The software has the following components:

- **Test Executive Software**
  Includes the complete functionality to define tests based on the provided block functions, to run tests and analyze the results.
  Type: R&S QS-APP, included option keys: R&S QS-EXE, R&S QS-EDI, R&S QS-SEQ
- **Test Sequencer Software**
  Executes QuickStep tests. This component is used in combination with a QuickStep application which provides the calling GUI.
  Type: R&S QS-SEQ, option key: R&S QS-SEQ
- **Development Option**
  Enables the creation of new block functions. The Block Development Tool provides the complete interface to integrate user code into QuickStep via MS Visual Studio projects.
  Type: R&S QS-DEV, option key: R&S QS-DEV
  Additionally required type: R&S QS-APP
- **OTA Basic Application**
  Provides the OTA ATSCAL view for easy and integrated control of OTA (over the air) RF radiation testing, particularly for calibration and antenna measurements in combination with an ATS1000. No configuration of a testplan or test procedure is needed. Test results (total radiated power, gain, radiation pattern) are also displayed in the view including polar and 3D chart.
  Type: R&S QS-ATSCAL, option key: R&S QS-ATSCAL
  Additionally required type: R&S QS-SEQ or R&S QS-APP
- **ATSDRV Positioner Driver Package**
  Provides the functionality to control an ATS-CCP1 antenna positioner with turntable and one antenna boom.
  Type: R&S QS-ATSDRV, free of charge

The licensing is realized with a **License Dongle** to be connected with a USB port at the PC where QuickStep is used. The license dongle consists of a smart card and a USB smart card reader. The definition and execution of tests with the Test Executive Software is limited to 25 steps if the license dongle is not available or the license is not valid. The development or OTA functionality is only available with valid license.

1.3 Documentation

PDF documentation

The pdf documents are included on the product's USB stick. Most documents are also accessible after QuickStep installation via the Windows "Start" button and the folder "R&S QuickStep > Documentation" or via the QuickStep Help menu. Additionally, the Getting Started is provided as printed document.

The pdf documentation consists of the following documents:

- **Getting Started**
  Provides basic information about the product and how to install it.
- **User Manual**
  Provides detailed information about the features of the application and how to install, configure and use the application. The manual includes descriptions of the applied mechanisms, step-by-step procedures showing how to carry out typical tasks, a reference chapter where the GUI elements and their usage are described, and application examples.
- **OTA Testing User Manual**
  User manual that is specialized for the OTA ATSCAL component of QuickStep.
- **Release Notes**
  Contains the most current information on the application, for example latest changes, news, restrictions.
- **Training Manuals**
  Provide detailed descriptions how to use QuickStep based on instructive examples. The descriptions include step-by-step procedures and many hints on practical usage.
  – User Training: Covers all topics related to the usage of QuickStep – except for the development of new blocks.

  – Developer Training: Covers the tasks for developing functional blocks. Code examples illustrating how user-defined block functions are developed. The example code can be copied and inserted in programming files in MS Visual Studio; therefore two versions of the training manual exist: one for programming in C++, one for programming in C#.

- **Quick Reference**
  Lists the typically required API functions on a poster.
- **ActiveReports User Guide**
  Describes how to use the ReportDesigner which is accessible via the Test Procedure Editor's toolbar. The ReportDesigner allows to create and edit report definitions and styles for the QuickStep reporting functionality.

Help, CHM documentation

- The context-sensitive help system is embedded in QuickStep. Press the "Help" button or the F1 key to access the help from the graphical user interface.
- The QuickStep API description is a help system describing the classes and files for block development. It is accessible via the Windows "Start" button and the folder "R&S QuickStep > Help and Manuals" or via the QuickStep Help menu.
- Developer documentation (CHM files) of the provided R&S base blocks for reuse of the block functions for the development of 3rd party blocks.

1.4 Typographic Conventions

This document uses the following typographic conventions to make information easier to access and understand.

Table 1-1: Typographic conventions

Convention	Description
"Reference" "GUI element" "Menu name > command"	"Quotation-marks" enclose references or graphical user interface citations to other documentation parts. The **>** symbol indicates a path or an order to follow for making selections on the GUI. Example: On the taskbar, click "Start > All Programs > ...".
[KEY]	[Capital] letters indicate key names. Example: CTRL key.

Convention	Description
Input	Letters in *italic* indicate a value you must type in as shown. Example: *5400*.
`code`	Letters in `fixed-width font` indicate `file names`, `commands`, or `program code`.
Link	Letters in blue font indicate links that you can follow (underlined). Links to the Glossaries are not underlined.
emphasis	Letters in boldface indicate **emphasized** words.
<variable>	Angle brackets enclose variable values. Example: <release>.
|	Vertical bar indicates alternate selections - the bar means "or".
...	Ellipsis indicates nonessential information omitted from the example.

1.5 Using the Help

The help system is integrated into the software. The help system is an integral part of the product's framework and thus provides information about all application parts that are officially released, independendly of whether the applications are installed or licensed, or not.

1.5.1 Accessing the Help

Use one of the following options to access the help. Depending on the specific help calls, several help windows may open in parallel.

- **F1 key**
  Opens the help. If the selected GUI element provides context-sensitive help, opens the related help topic.
  For example, if a tab is selected, F1 will open a reference description for the tab.
- **Menu bar**
  The "Help > R&S QuickStep Help" item opens the Online help ("Help" provides links to other documents, too).

Using ToolTips
Move the pointer over an element to display a short description of it.

1.5.2 Navigating in the Help

As in most help systems, you can use common tabs to find the information of interest. You find the tabs in the left pane: "Contents", "Index" and "Search". The icons in the toolbar provide further navigation support, see the following table.

Table 1-2: Help menu

Command	Description
"Hide"/"Show"	Hides or shows the left pane.
"Previous"	Opens the previous page of the help directory.
"Next"	Opens the next page of the help directory.
"Back"	Opens the topics you visited before.
"Home"	Shows the start page of the help (legal notice).
"Print"	Lets you print the current topic or the selected heading and all subtopics.

2 What's New

This document is related to **version 4.65** of QuickStep and later.

No new features are described in the document since version 4.50. For more details, see the Release Notes.

3 Introduction to QuickStep

This chapter provides a brief overview over QuickStep for a first orientation. The given information is not comprehensive and not represented with full complexity.

3.1 Typical Test Setup

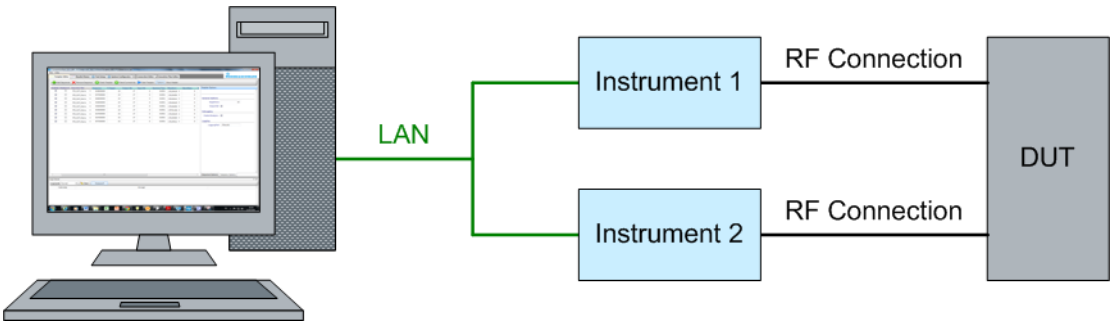


Figure 3-1: Schematic test setup (DUT: Device under Test)

Characteristics:

- QuickStep runs on a PC and controls the test instruments.
- QuickStep basically commands a sequence of test steps where the values of one or several test parameters are varied. The results for each test step are collected and presented within QuickStep.
- Typically, SCPI commands sent over LAN (or GPIB) control the test instruments. Any other remote control interface might be adapted.
- The test instruments can be of any type. Examples are generators, analyzers, power supplies, power sensors, switching devices. The number of used test instruments is not limited.
- One or more test instruments provide test signals as input for the DUT. Vice versa, one or several test instruments gather signals or data from the DUT.

  Examples:

  – A generator instrument provides an RF signal to the DUT. QuickStep defines the properties of the RF signal to be transmitted.
  – An analyzer instrument receives RF signals from the DUT and measures their properties. QuickStep gets the results from the analyzer.
  – A power supply with variable voltage powers the DUT.

3.2 Graphical User Interface

All operational tasks for configuring and executing tests are carried out on the PC. When starting QuickStep, the "QuickStep" window – the graphical user interface (GUI) – opens.

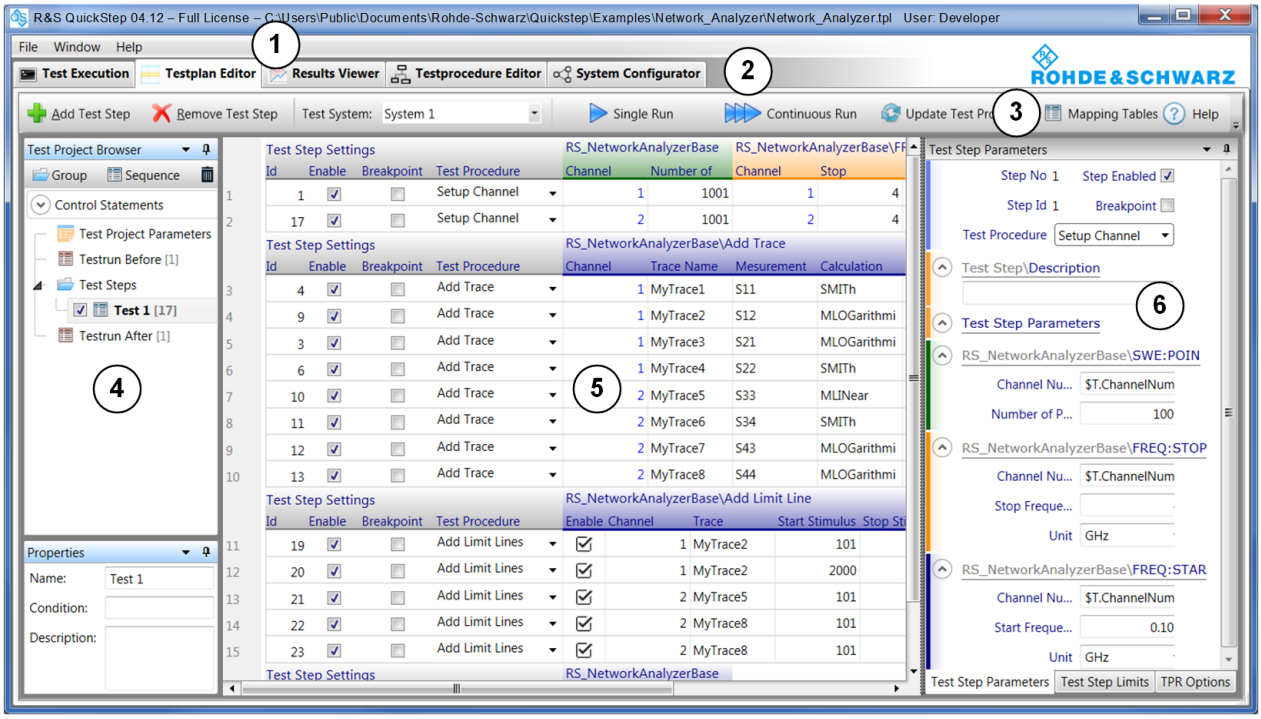


Figure 3-2: GUI overview

1 = Menu bar
2 = Tabs
3 = Toolbar
4 = Navigation / browser / library
5 = Main pane
6 = Secondary pane

The GUI is structured with a menu bar, tabs, a toolbar and several panes. The content to be displayed is distributed in several tabs. The selected tab defines which type of information is displayed in the different panes. See the descriptions below for information on the content for single tabs. The entries in the toolbar also depend on the selected tab.

3.2.1 Test Execution

This view becomes relevant when the current test is executed. You can start the test run and control the test execution.

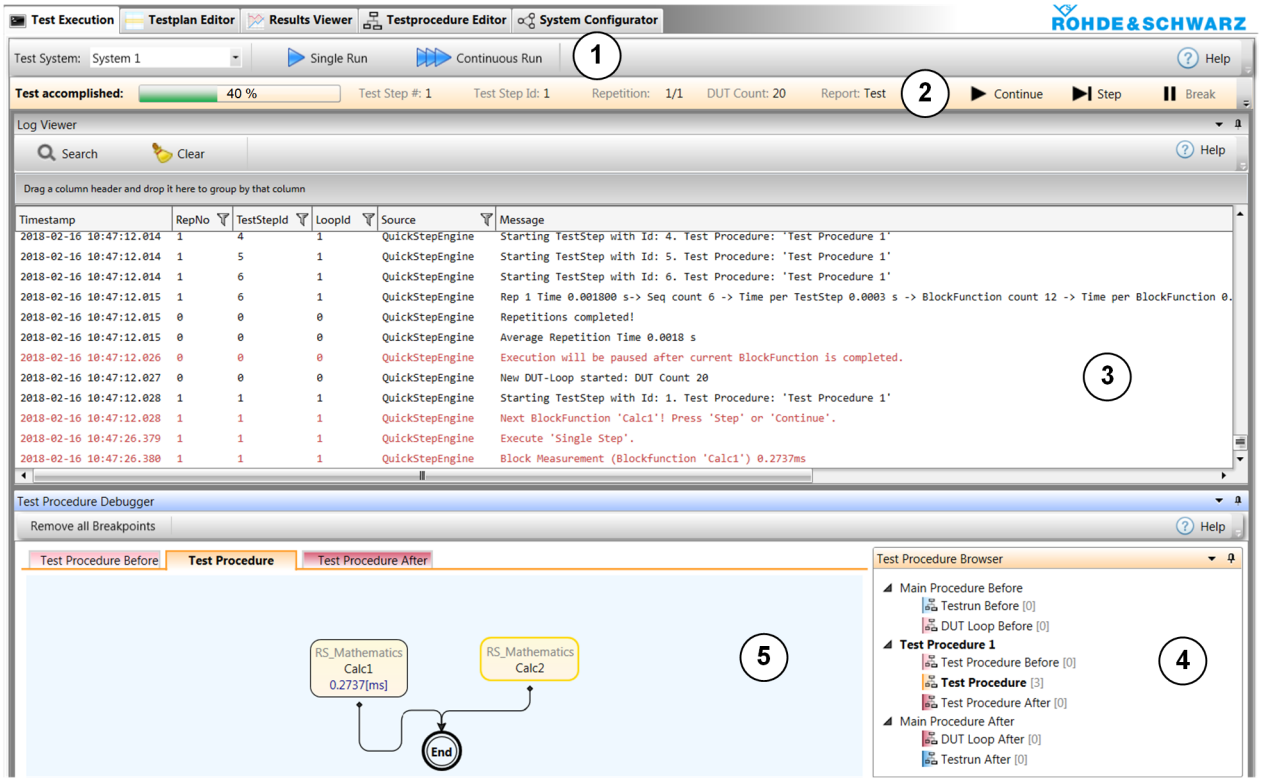


Figure 3-3: Test Execution

1 = Start the test execution
2 = See and control the execution progress
3 = View the logged messages
4 = Select the block function flow chart of interest
5 = Inspect the current block function

Progress bar

The progress bar shows how far the test has been executed. You can control test execution, for example resume test execution after a halt due to a breakpoint in the test plan.

Log Viewer

The Log Viewer protocols the events occurred during operation of QuickStep, particularly after starting the test execution. The messages are color-coded.

Test Procedure Debugger

The "Test Procedure Debugger" allows to check the values of parameters during test run. It includes the Test Procedure Browser from the Test Procedure Editor for selecting the block function diagram that contains the block function of interest.

The debugger works together with the progress bar during test run. You can proceed in the test execution step by step with the "Step" button. If you have defined breakpoints for test steps (to be done in the Testplan Editor) and have clicked the "Continue" button, the test execution is halted at each breakpoint until you click "Continue" again.

3.2.2 Test Plan Editor

The "Test Plan Editor" is the initial view of QuickStep. The user prepares a list of test steps and starts the test execution from the toolbar.

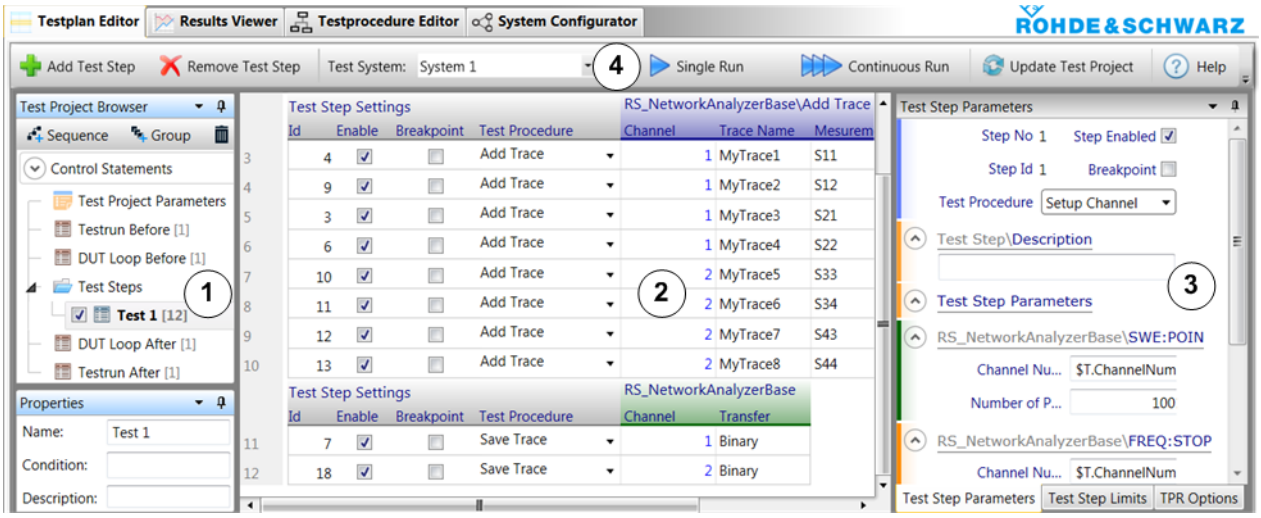


Figure 3-4: Testplan Editor

1 = Select a sequence of test steps (or define groups and sequences)
2 = Inspect and edit the sequence of test steps
3 = Edit parameter values
4 = Start the test plan

Central test step table

In the table, each test step is represented in one row, the columns display the related parameters. Parameter values can directly be edited in the table after a double-click.

Each test step is connected to a test procedure by the entry in the "Test Procedure" column. The parameter set of each test step is dynamically adapted according to the selected test procedure. If test procedure parameters are modified in the test procedure editor, the modifications get effective in the test plan editor after clicking "Update Test Project".

Powerful sweep and set functions allow quick generation of parameter sweeps for efficient parameter setting of multiple test steps. Multi-parameter sweeps might be defined within one single test step. Prioritization might be used to keep control on the order of the parameter sweeps within nested loops.

Panes on the right-hand side

In the "Test Step Parameters" tab, the parameters of a test step are displayed in vertical order for a better overview and providing a more convenient way to edit parameters without scrolling. The "TPR Options" tab contains parameters for the whole test, for example repetitions. The "Test Step Limits" tab shows the configured limits for measurement results.

Regarding test development, various settings for logging and debugging are offered. Breakpoints for debugging and single-step execution can be enabled for specific test steps.

Test Project Browser on the left-hand side

Multiple test step parameter tables are organized in a tree structure for keeping an overview of large tests. Control structures can be applied to sequences of test steps, their parameters are configured in the right pane. "Test Project Parameters" contains static and dynamic global parameters to be configured in the middle pane.

3.2.3 Results Viewer

The results of a test run are displayed in the "Results Viewer".

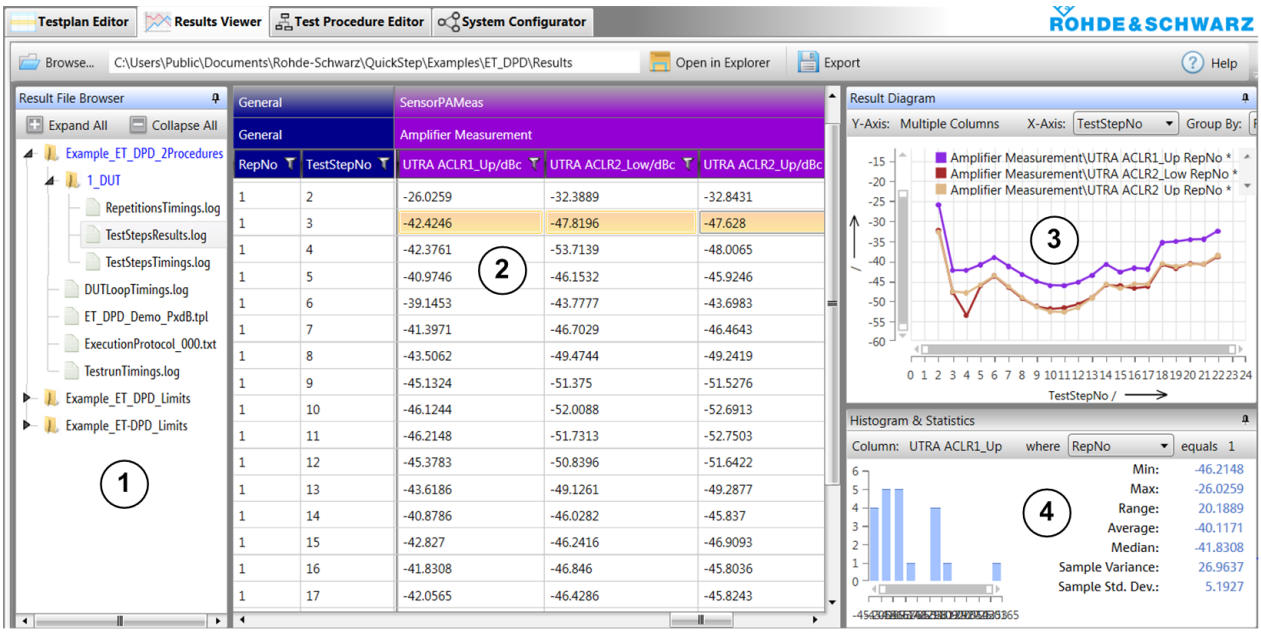


Figure 3-5: Results Viewer

1 = Select a result file
2 = Inspect the result table and select one or more result columns
3 = Inspect the diagram representation of the results for the selected column(s)
4 = Inspect the distribution of result values and check the statistical evaluation

Results File Browser on the left-hand side

The "Result File Browser" helps to keep an overview of large sets of result data. Each test run generates a new time-stamped folder with a complete set of result files with measurement and timing results as comma-separated value (CSV) files. For each DUT, a separate subfolder is created. If results of the type trace are generated, these are also collected in a DUT-specific subfolder. Additionally, a copy of the test plan and the execution log is stored as a reference.

When selecting a result file in the Result File Browser, its content is shown as table in the central area. `TestStepsResults.log` is the main result file containing the results for each test step. `ExecutionProtocol ... .txt` contains all logged messages with timestamp and origin.

Central Results Table

The central "Results Table" shows the results in a table. In case `TestStepsResults.log` has been selected, each test step is presented in one row and each result parameter in one column. If one or several result parameters

in the results table are selected, the results over the test steps (or other configurable running variables) are represented in the diagram on the right-hand side.

Each column of the table offers powerful sort and filter functions. An export filter makes it possible to export a subset of the table as CSV or XLS file. In case the table shows the content of the execution protocol, it is possible to export and reuse SCPI sequences within other test environments (for example).

Analysis panes on the right-hand side

The "Diagram" pane plots the data of a single or multiple columns that are selected within the result table. Scatter plots are possible, since any result parameter can be selected for the x-axis of the plot. Results can be assigned to color-coded groups by selecting an additional grouping parameter. Delta markers are available for measurements.

If results of the trace type are selected, an adopted diagram pane is available. Traces files can also be loaded directly into the central results table and displayed with the standard results viewer. Zoom in and out is supported by mouse click, mouse wheel and diagram bars.

The "Histogram & Statistics" pane provides a histogram pane and statistical analysis of the result data that is selected within the result table.

3.2.4 Test Procedure Editor

A test procedure basically defines what functionality is executed when the test steps connected to the test procedure are carried out. It is set up as flowchart with a graphical editor, based on a library of provided functions or user-developed functions.

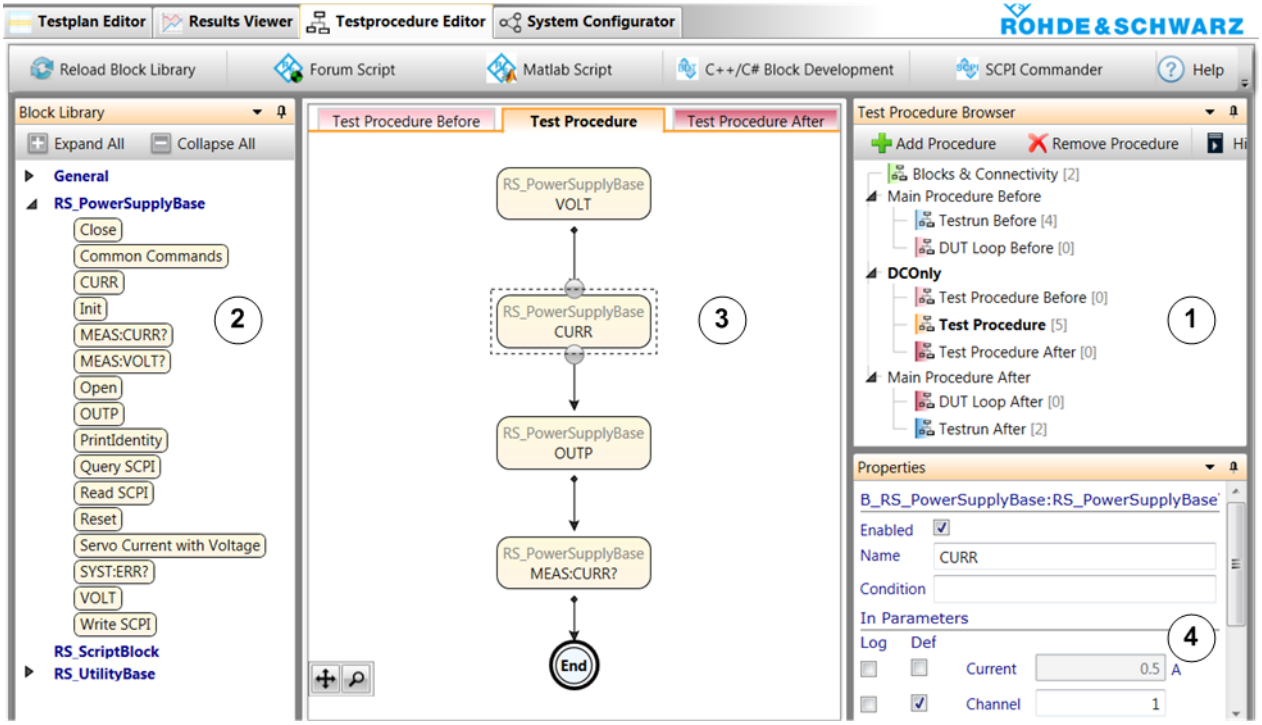


Figure 3-6: Test Procedure Editor

1 = Select an execution phase
2 = Drag a new block function into the main pane
3 = Select block functions, add block function dependencies, select a block function
4 = Edit the parameters of the selected block function

Control elements such as "If", "Or", “Fork” and “Join” are available to handle execution branches and loops. Conditions achieve a conditional execution of test functions. All test function parameters can be made available for test parameterization within the test plan editor. Existing test procedures can be modified or extended without source code development.

The toolbar provides access to tools for developing blocks, handling SCPI commands for connected test instruments, designing reports and integrating scripts.

3.2.5 System Configurator

The "System Configurator" reflects the test setup and can be used for setting the device- and connection-specific parameters as occurring with the test setup.

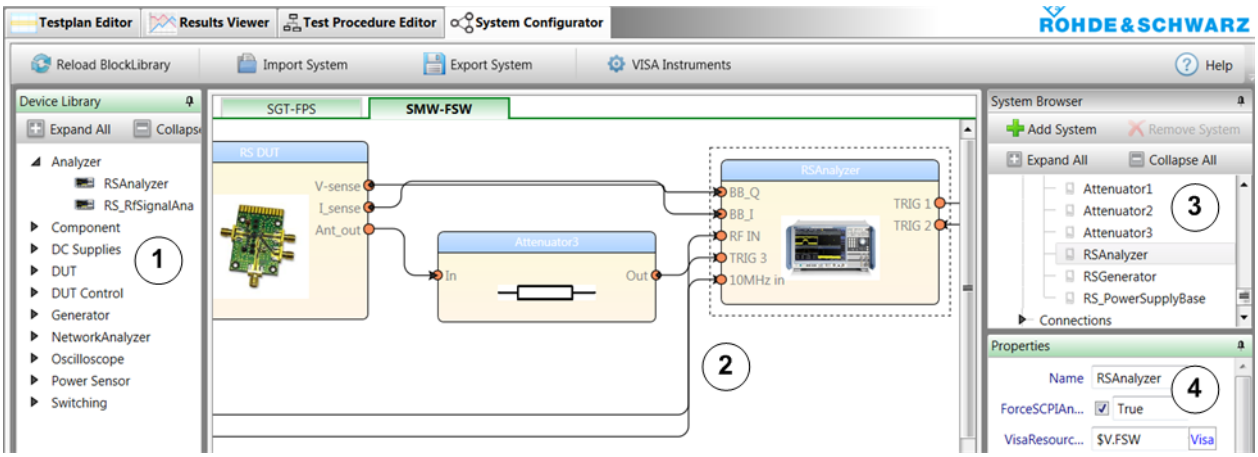


Figure 3-7: System Configurator

1 = Drag a symbol into the main area
2 = Connect the symbols
3 = Select an element
4 = Edit the properties (parameters) of the selected element

The main pane displays the used devices (test instruments, components, even attenuators) and connections. You drag devices from the "Device Library" on the left pane into the main pane. Then you draw connections between the devices. On the right side in the "Properties" pane, you can see and edit the properties of the currently activated device.

The system configurator facilitates the handling of several use cases:

- Assistance for building up a VISA connection to a test instrument.
- Automatic calculation of the RF path loss during test execution. Attenuations/ losses for individual components of the system are defined, then one or more connections and system components are assigned to an RF path.
- Easy switching between several test benches. Therefore, the system configuration contains the configuration for each of them.
- Management of system-dependent parameters like connection IDs.

3.3 Block Function Development

The Block Development Tool is provided for defining new test blocks, test functions and the associated function parameters. Based on these definitions, Microsoft Visual Studio C++ or C# projects with source code templates are automatically generated. The templates just have to be extended with user code in order

to create user-specific test functions. The newly developed test functions are available in the test procedure editor after compilation.

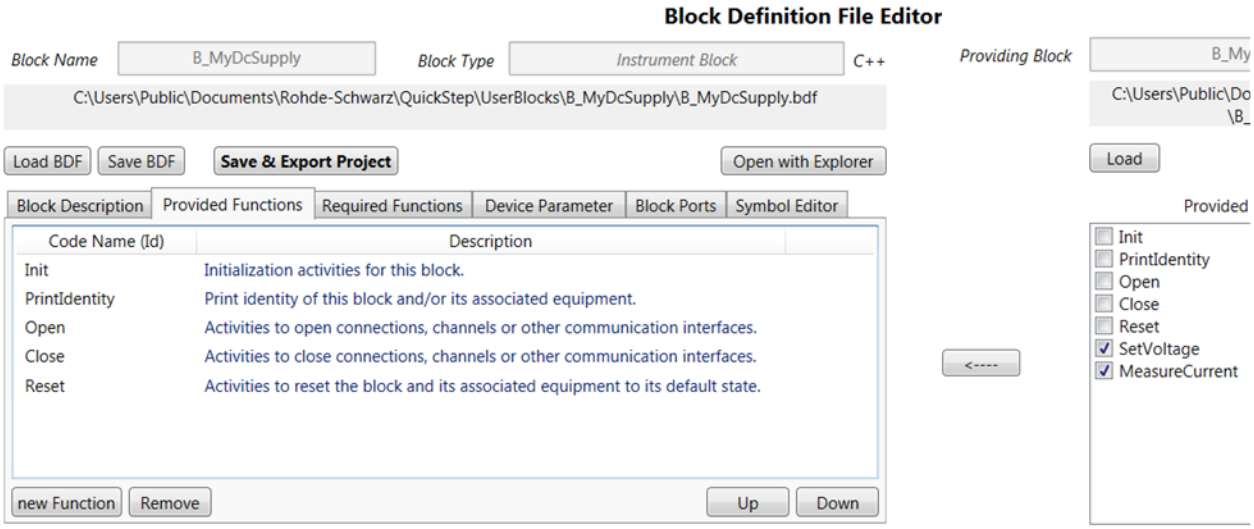


Figure 3-8: Main section of the Block Development Tool

The QuickStep API (application programmer interface) offers a set of functions for data exchange with other functions and logging of results. Even users with limited software development experience can implement new test functions with just a few lines of code. Development experts can exploit all capabilities of Visual Studio for development of complex test functions.

4 Concepts

4.1 Test Structures and Their Relations

Test plans, test procedures and blocks

QuickStep organizes a test configuration with the following main structures:

- **Test plan**: Contains a sequence of test steps, each one using a set of parameter values, and scheduling information. Each test step is connected with a test procedure. Multiple test procedures for different test purposes may be used in one test plan.
- **Test procedure**: Defines the test functionality to be applied within a test step. A test procedure is structured by a flowchart of block functions (see below) which execute application software. The block functions can be arranged with certain control structures.
- **Block**: Contains a set of functions and one function is selected. A function fulfills a certain task and provides the parameters for user input. Often, a block reflects the functionality of a test instrument. But it can also represent instrument-independent functions ("software block") or only parts of an instrument's functionality.

This structure keeps the test steps free from the real test equipment functionality. The concept of blocks carrying functions allows you to provide, adapt and develop test algorithms optimized for special test equipment and test conditions. The blocks in a test procedure can be arranged in parallel or in sequence or in combination with decision functions.

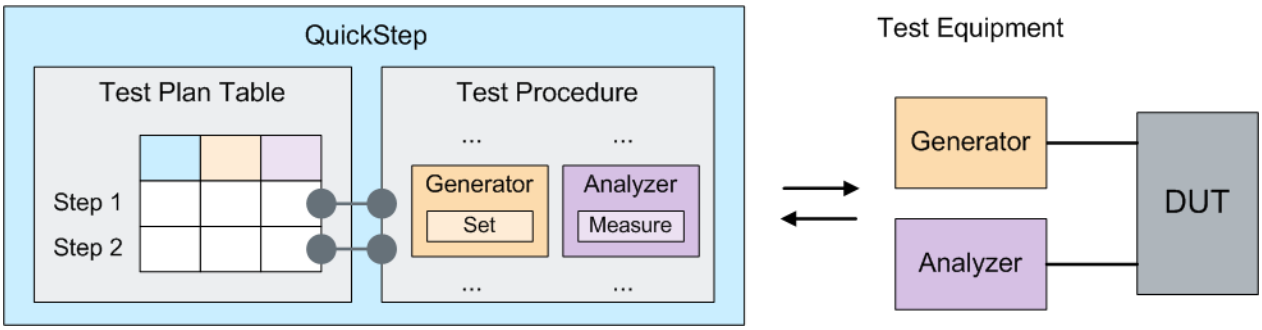


Figure 4-1: Logical representation of a test (principle)

A test step is connected with a test procedure by its Test Procedure value in the testplan table. See the figure where test steps 1 and 2 are connected to Test Procedure A.

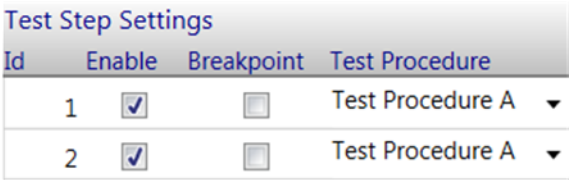


Figure 4-2: Connecting a test step with a test procedure

Parameters

Parameters provide the concrete, quantitative settings used for the test execution, for example frequency and power values.

The same parameters are related to block functions, test procedures and test plans as well: The parameters required for specific functionality (for example for controlling a generator instrument) are defined by block functions. So, they appear in a test procedure when the related block function is added in the procedure. Afterwards, they get available for the test steps of a test plan table when the test steps are connected to the test procedure.

In the test plan table, the parameters are displayed as columns, grouped by the originating functions. Usually, the parameter values are set in the test steps of the test plan with certain variations over the test steps. When the test is executed, the functions of the connected test procedure take over these values e.g. for their communication with the test instruments and calculations. This interplay between test steps and test procedure is fundamental; a test step without connected test procedure is undefined.

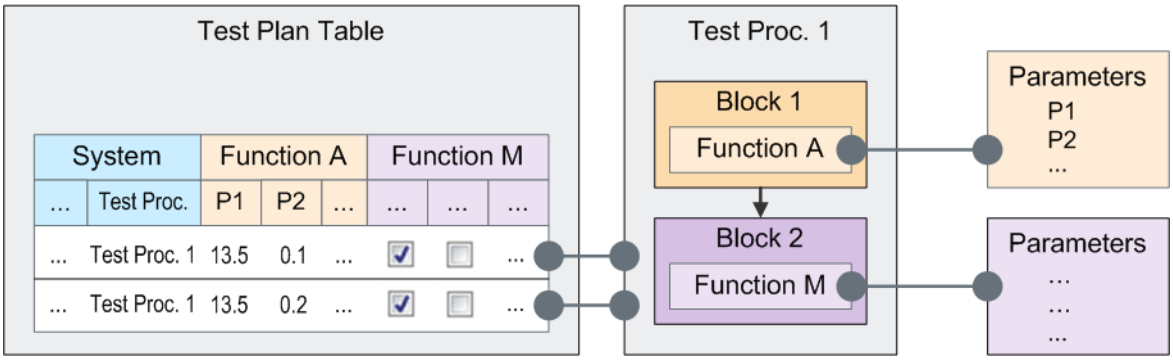


Figure 4-3: Definition and display of parameters (principle)

The default values of the parameters in the test plan refer to the parameter values set in the "Properties" section in the "Test Procedure Editor". Parameter values can also be set by references, see chapter Setting Parameter Values by References.

Additionally, common parameters for a test procedure or even the test project can be defined.

Test run mechanism

When starting the test execution, the test steps of the test plan are processed according to the order and the repetitions defined in the test plan. For each test step reached during the test run, the connected test procedure is called. The test procedure takes over parameter values from the test step and executes the functions as defined in its blocks and regarding their arrangement.

The sequence of test procedure executions for the test steps is enframed with some procedure-like control structures (for example "Testrun Before"). Usually, the term "Test Procedure" is reserved for the procedures related to test steps only, but if convenient, the additional procedure structures can also be included. The parameter sets of the control structures also appear around a test step table and are included in the term "Test Plan" if convenient.

4.2 Test Project

A test project is the master structure for the components of a QuickStep test. For each test, it contains the test plan and test procedures as well as result and log files, various configuration files and other components. The following figure gives a symbolic overview over the main components, their main substructures and relations. The blue colored fields correspond with the main GUI partitions (Test Plan Editor, Results Viewer, Test Procedure Editor and System Configurator). The system configuration is optional.

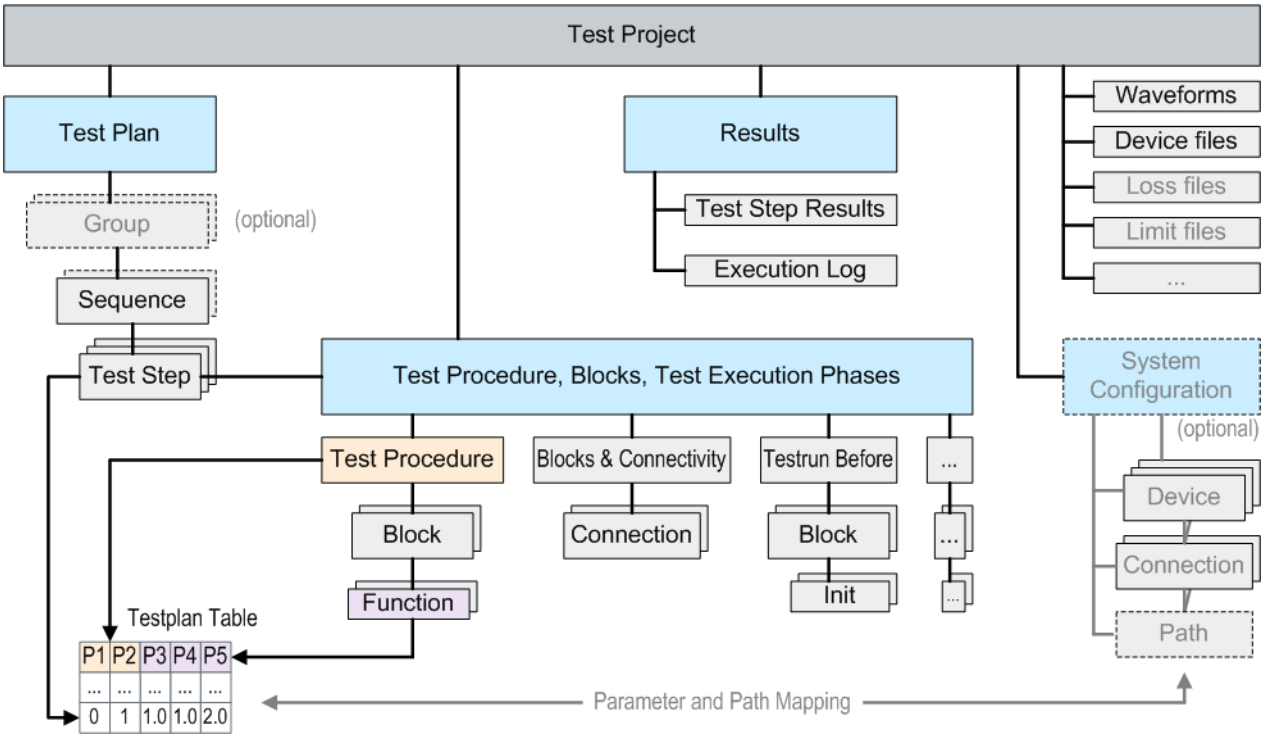


Figure 4-4: Test project structures and relations (simplified)

See Chapter A, "File Extensions and File Locations", on page 288 for information about actual file locations.

4.3 Test Execution Phases

An overall test run consists of several phases. Phases which contain block functions are displayed in the "Test Project Browser" within the "Testplan Editor". The parameters for the currently active phase are displayed in the table area of the "Testplan Editor". The functionality to be executed for such an item is configured by block functions in the "Test Procedure Editor".

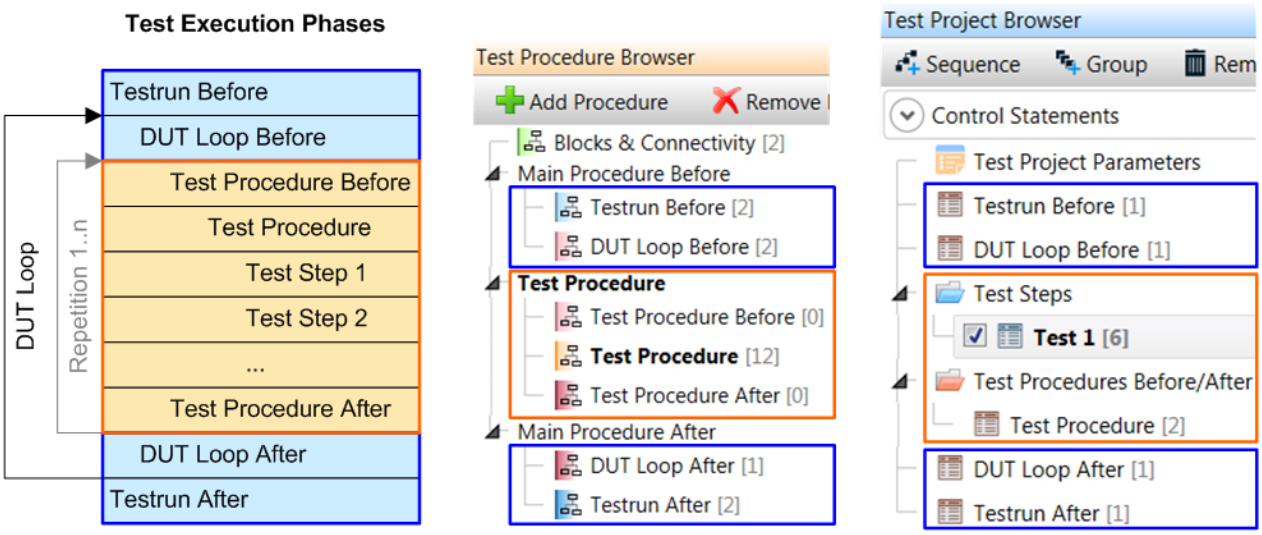


Figure 4-5: Test execution phases and their representation in Testplan Editor and Test Procedure Editor

The "Test Steps" phase is the main one where the actual testing takes place. Other phases control settings before the "Test Step" phase is started or after it has ended.

Test execution phases:

- "Testrun Before" and "Testrun After" (mandatory phases): are executed once at the beginning and end of a test run. They are used for initializing the test instruments and for closing their activities and the connections to them in the end.
- "DUT Loop Before" and "DUT Loop After" (optional, not displayed here if no function is connected) become relevant if several DUTs are tested in a row (DUT: Device under Test). They are executed at the begin and end of the test of each DUT. For details, see Chapter 4.10, "DUT Handling", on page 62. The DUT loop mechanism can be used for the control of a handler in a production environment.
- "Test Procedure Before" and "Test Procedure After" (optional) become relevant if test steps of the test plan are assigned to different test procedures. The functions within these phases are executed whenever the test procedure changes during test run. Use these phases for setting and assuring defined system states when changing the test procedure.
- "Test Procedure" (mandatory) executes the sequence of test steps as defined in the table of test steps.

Repetitions of the test procedure execution phases can be used to validate the stability of measurement results and to improve the reliability of results. Result data is generated in a loop for statistical analysis.

4.4 Test Plans

A test plan consists of one or more test sequences, each with individual test steps to be executed consecutively. Several sequences can be collected in a group and the test plan can comprise several groups. The test plan also contains scheduling information such as the number of repetitions and conditions for the execution of a group or sequence of test steps. Each test step is represented by a row in the test plan table. The test steps have groups of parameters, see the following figure showing an example. Each group to the right of the Test Step Settings and Test Procedure Parameters represents the parameters of one block function.

	Test Step Settings				Test Procedure Parameters			Generator Function Parameters			Analyzer Fct. Param.		
#	Id	Enabled	...	Test Procedure	Frequency	...	...	Target P.	...	Waveform	ACLR	...	EVM
1	1	☑		Test Proc. 1	3510000000					LTE10M50RB ▾	☑		☐
2	2	☑		Test Proc. 1	3520000000					LTE10M50RB ▾	☑		☐
3	3	☑		Test Proc. 1	3530000000					LTE10M50RB ▾	☑		☐
	...			...	...					...		...	

Figure 4-6: Groups of test plan parameters (example)

- The Test Step Settings (light blue) comprise some control parameters. The Test Procedure column offers a drop-down menu for selecting the test procedure used for this test step (if more than one test procedure has been defined within the Test Procedure Editor).
- The Test Procedure Parameters comprise the parameters with common settings for several block functions. The block functions take over the parameter values by reference. This mechanism avoids that common parameters have to be set at the generator and the analyzer separately).
- The parameters of the generator function (selected in a generator-related block) define the characteristics of the signals which the generator transmits to the DUT, for example frequency and power. The shape and details of the signals are determined by waveform files.

- The parameters of the analyzer function (selected in an analyzer-related block) define which measurements are executed, for example the Adjacent Channel Leakage Ratio (ACLR) or the Error Vector Magnitude (EVM).

It depends on the selected test procedure and its underlying functionality which parameter columns are displayed (dynamic column sets).

4.4.1 Test Sequence with Several Test Procedures

The following figure shows the execution order of the phases for a test run where two test procedures are involved.

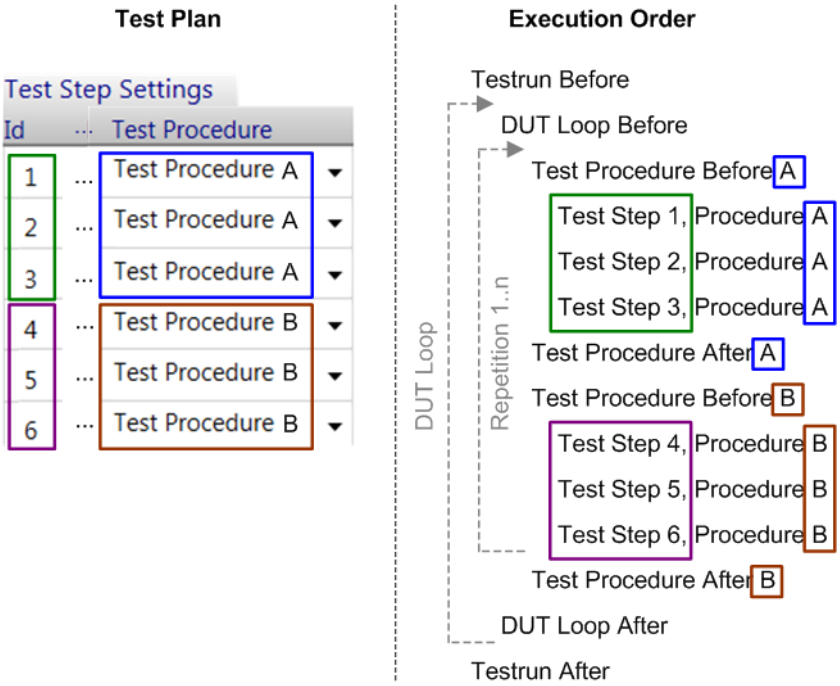


Figure 4-7: Test with two test procedures

4.4.2 Single-Line Sweep

A parameter in the test plan table can be varied by a loop within one test step. This mechanism is called single-line sweep. It realizes a sequence of value iterations for a parameter in a compact way while the other parameters keep their values. The single-line sweep belongs to one table cell (determining both the parameter and the test step) and is defined by usual loop parameters: Start value, step size and count (or stop value instead of step size). Such a sweep can conveniently be set up with the help of the "Edit Test Step" dialog from the context menu.

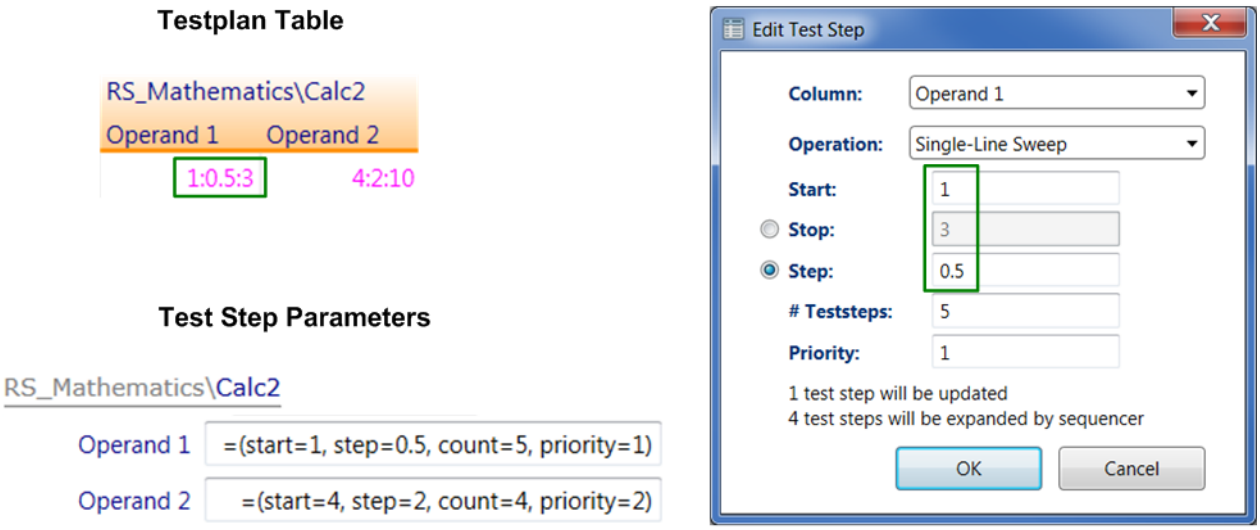


Figure 4-8: Definition of a single-line sweep

For each loop iteration, the complete test step actions are carried out and an own result is logged. The results are distinguished by their loop ID in the "Loop ID" column of the results table.

If several single-line sweeps are applied in the same test step, the loops are nested. A priority parameter determines the order of the loops: Assumed that there are two single-line sweeps (for example with priority values 1 and 2), the sweep with the higher priority number (2) is carried out in the inner loop. In case of several single-line sweeps with same priority, all parameters are altered in each test step.

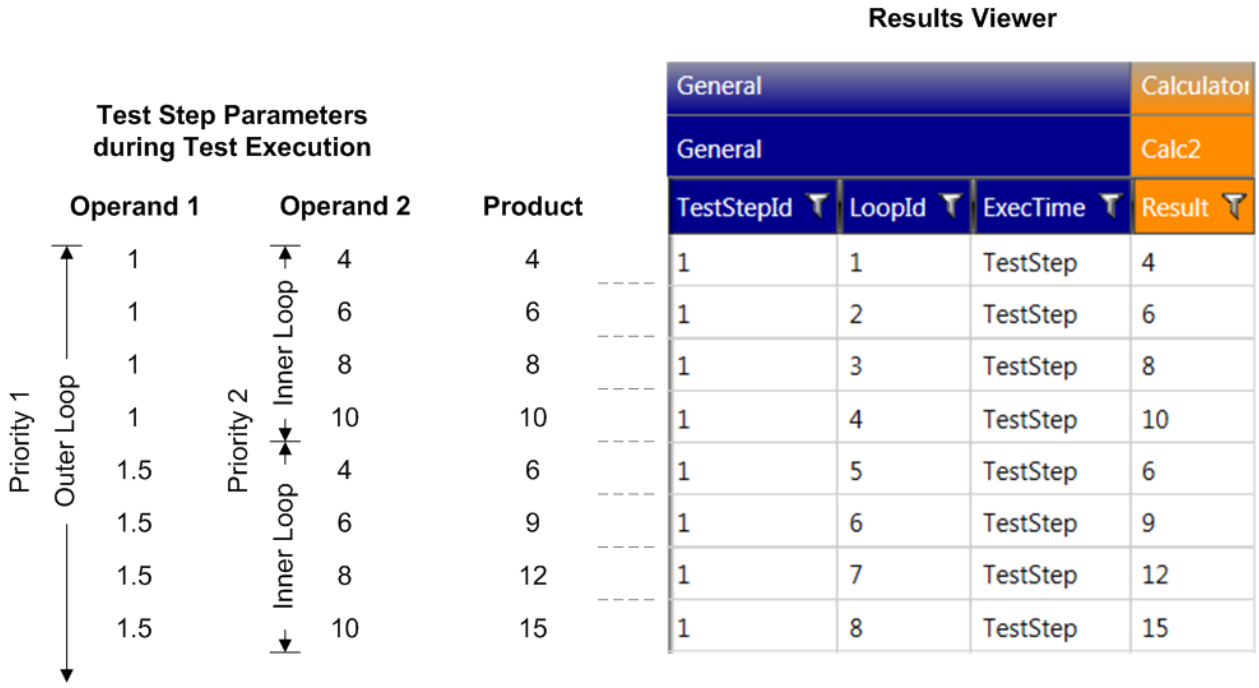


Figure 4-9: Execution and result of a single-line sweep

4.5 Test Procedures

Test procedures control the operation of the test instruments and are structured by blocks and selected functions – short: block functions. Dependencies between block functions and conditions for their execution can be defined. The following figure shows a part of an example test procedure. The test procedure is related to the "Test Procedure" item of the "Test Procedure Browser". "Test Procedure" contains the functionality to be executed for each test step which is connected to that procedure.

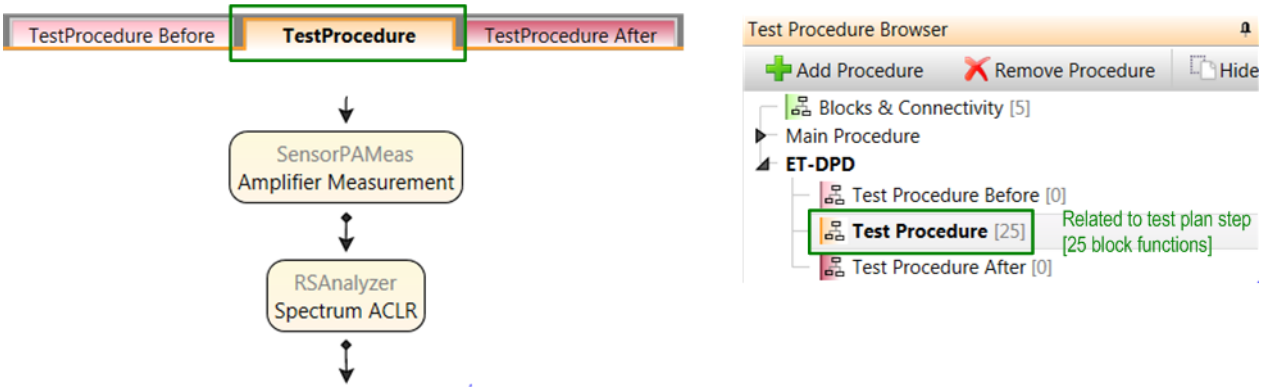


Figure 4-10: Test procedure (example)

Regarding a complete test run, a test procedure is embedded in some procedure-like control structures. The overall functionality for a test run is distributed in a "Blocks & Connectivity" section and several test execution phases (for example "Testrun Before"; "Test Procedure" is one of them). The phases are selectable via the **"Test Procedure Browser"** on the right side of the Test Procedure Editor.

4.5.1 Blocks & Connectivity

All blocks which are used within the test procedures of a test project have to be added to "Blocks & Connectivity". If a block is not contained in "Blocks & Connectivity", it is not available for any test execution phase. Several instances of a block can be used.

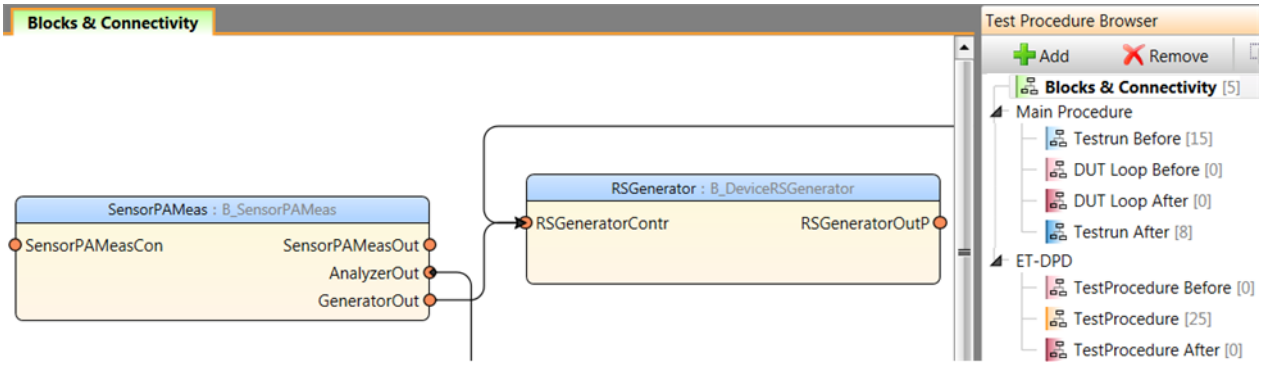


Figure 4-11: Blocks & Connectivity (example)

Block Connections

Connections between blocks define the paths for inter-block communication (data transmission between blocks). Such block connections are created and displayed in the "Blocks & Connectivity" tab. Block connections are required if a block

needs information from another block, for example state or control information or any kind of data, to work properly. Or a block could require another block to execute a specific function.

Each block connection is represented by an arrow line from an out port to an in port. The connection has its origin at an appropriate out port of the block requesting some data or activity. It terminates at an appropriate in port of the block receiving the request. The answer to a request is sent back over the same connection.

The blocks provided within QuickStep already indicate by their ports if they can have a connection for passing or getting information. A port name often tells which type of block is needed for the other end of the connection. For example, the GeneratorOut port of the requesting SensorPAMeas block is connected with an in port of the receiving RSGenerator block (see the figure) which is the RSGeneratorContr port.

When developing a new block, its block ports have to be defined via the Block Development Tool, see for example Chapter 9.2.2, "Block Definition File Editor", on page 256.

4.5.2 More about Test Execution Phases

The test execution phases are selected in the "Test Procedure Browser". The functionality of the different test execution phases is defined by flow charts of block functions in the main (middle area) of the Test Procedure Editor. Each phase has its own block functions.

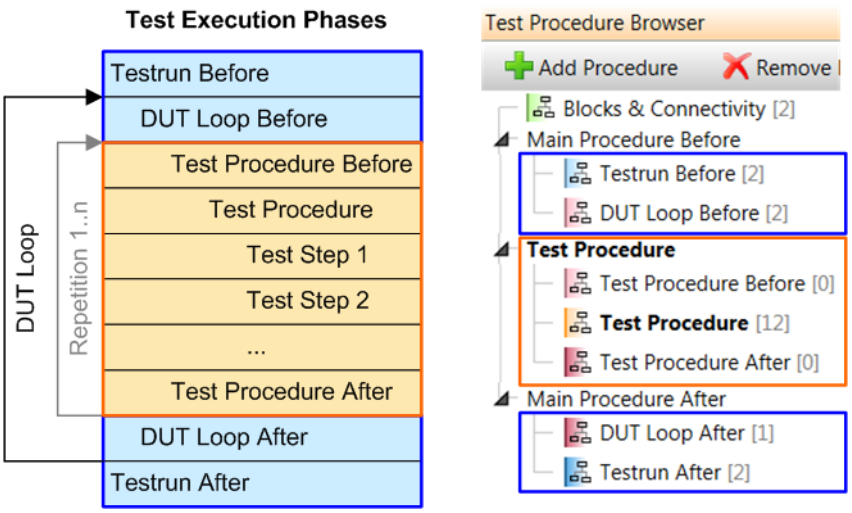


Figure 4-12: Test execution phases in the Test Procedure Browser

The "Main Procedure ..." nodes within the "Test Procedure Browser" collect the phases which serve as frame for the test run. The node following "Main Procedure Before" contains the actual test procedure. Each "Test Procedure" has two enframing phases "Test Procedure Before" and "Test Procedure After".

Several test procedures used in one test plan

Several test procedures (with accompanying "Test Procedure Before" and "Test Procedure After" phases) can be defined in the "Test Procedure Browser". A test plan may use more than one of them. The "Main Procedure ..." actions are applied for all test procedures.

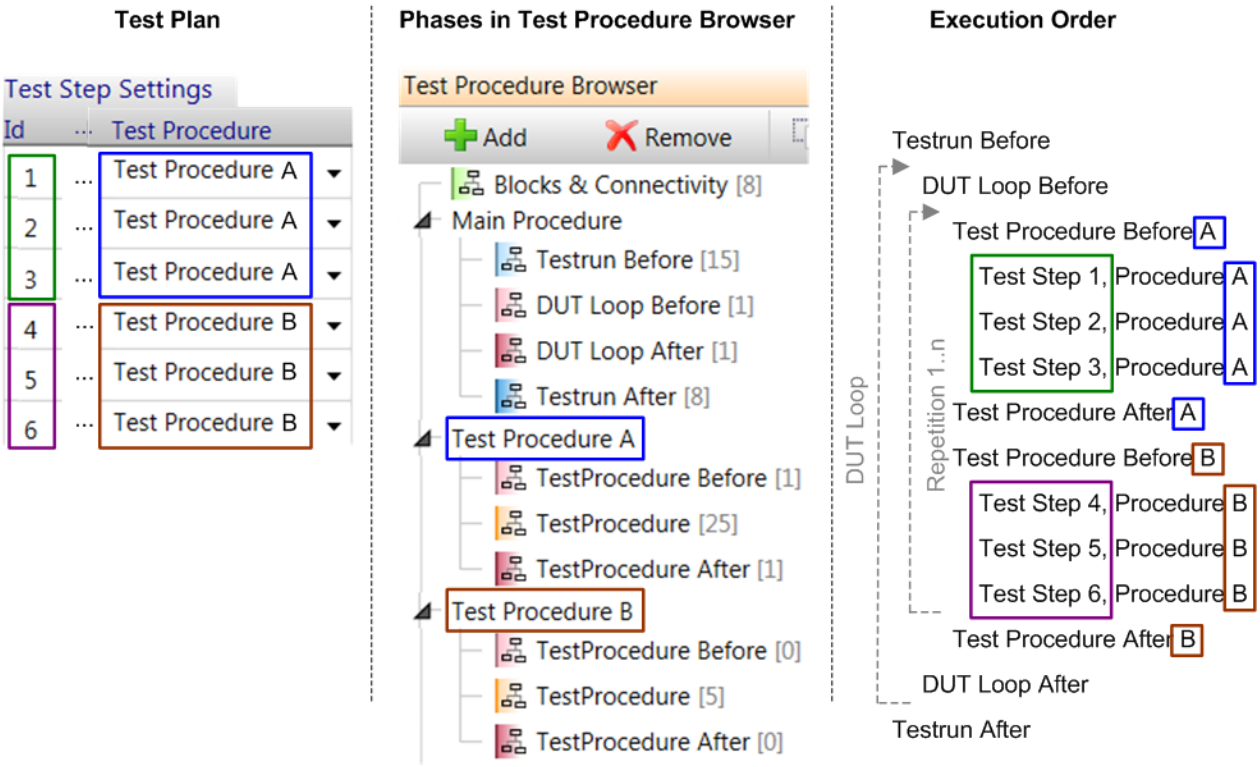


Figure 4-13: Several test procedures used in one test plan

4.5.3 Block Functions

A block is displayed with its name given in the headline and a selected function below the block name. The function defines what is done when the block comes into action. Generally, the functions operate based on associated parameters which are displayed in the "Properties" area on the right side of the Test Procedure Editor. The parameter values are often set in the current test step of the con-

nected test plan. A parameter appears in the test plan table and is set there if its check box in the "Properties" area is not ticked.

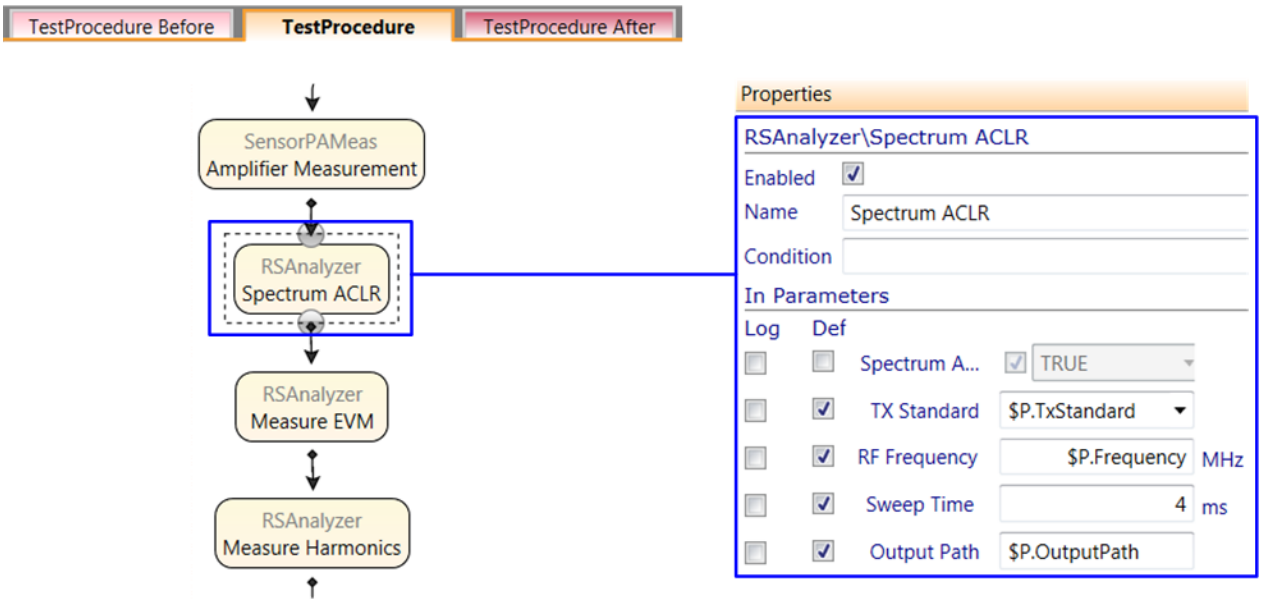


Figure 4-14: Parameters associated with a block function

Rohde & Schwarz provides blocks for several R&S test instruments. You can also create your own blocks and integrate any functional code, see Chapter 7, "Block Development", on page 117. In all cases, the block structures are the same.

4.5.3.1 Execution Conditions

Each block function in a test procedure contains the "Condition" property which controls if the block function is executed or not depending on the values of parameters during test run. The condition is a functional and logical expression with parameter values, mathematical functions and operators and logical operators. It is evaluated when the block is reached during test run. If the condition is satisfied (true), the block function is executed. Else (false), the block function is skipped. Blocks are displayed in blue color in the test procedure editor if they contain a dependency.

Conditions can contain references to parameter values from other data structures (test plan, test procedure, test result), see Chapter 4.7, "Setting Parameter Values by References", on page 49 and Chapter 6.4, "Setting Values by Reference", on page 97.

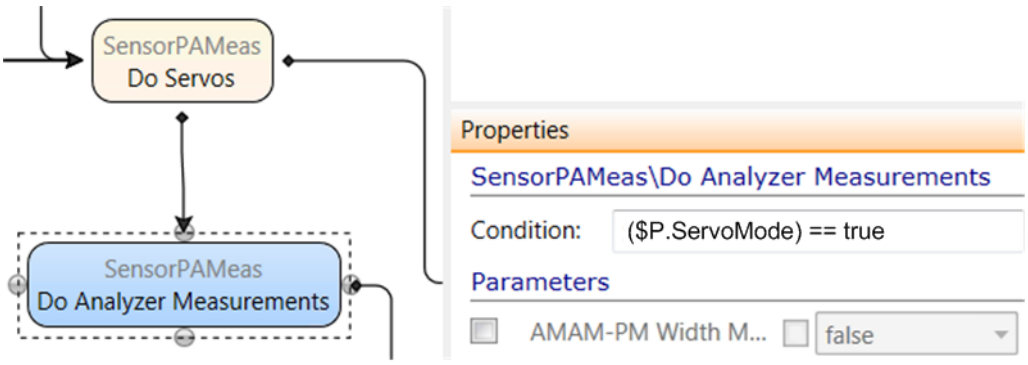


Figure 4-15: Execution condition for a block function

Syntax rules:

- The condition has to be expressed in accordance with the Mathematical Expression Toolkit Library (ExprTk) which QuickStep uses for evaluation.
- Example elements:
  - Mathematical operators: +, -, *, /, %, ^
  - Functions: min, max, avg, sum, abs, ceil, floor, round, roundn, exp, log, log10, logn, root, sqrt, inrange)
  - Equalities, Inequalities: =, ==, !=, <=, >=, <, >
  - Boolean logic: and, mand, mor, nand, nor, not, or, xor, xnor
- References must be enclosed in brackets.

Examples:

- Condition: ($P.Frequency) >= 1900
  The block function is executed if the value of the test procedure ($P) parameter "Frequency" is equal to or exceeds 1900.
- Condition: ($T.ServoMode) == true
  The block function is executed if the value of the test project ($T) parameter "ServoMode" is true.
  This kind of condition is useful for controlling a fork block connection, e.g. for two fork ends: The block function at the left fork end gets the condition ($T.ServoMode) == true, the block function at the right fork end gets the condition ($T.ServoMode) == false.
- Condition: ($R.<RS_Mathematics><Calculation><Result><this><this><this>) <= 16
  The block function is executed if the current value of the Result parameter is equal to or less than 16.

Note that a condition containing a result reference is resolved at runtime during test execution. The affected block function is not activated until the result is available. Depending on the value of the expression in the condition, the block function is then either executed or skipped.

For more details and examples using references and expressions in conditions, see the QuickStep User Training manual.

4.5.4 Connections between Block Functions, Control Structures

Arrow connections in a flow chart of blocks define the execution order within a test procedure phase. They are drawn via mouse. First, the block function at the arrow base is processed, then the system proceeds to the block function at the arrow head. All root block functions without incoming arrow are initially started in parallel when the test procedure is executed. Block functions belonging to different blocks are executed in individual Windows threads.

Fork and join dependencies

Blocks without dependencies are executed in parallel. Explicit parallel execution of functions is supported with fork and join dependencies between several blocks. These dependencies can be drawn in the "Test Procedure Editor" arrow line by arrow line in the same way as simple dependencies (via mouse). The additional Fork/Join element provides a simultaneous forking and joining point for any number of incoming and outgoing dependencies. Even a combined join and fork structure can be realized (e.g. three incoming block dependencies and two outgoing dependencies). The Fork/Join element improves the readability of test procedures in case of forks and joins but has no other function (forks and joins can also be realized without this element). It is transparent for the test execution, and the execution timing is already determined by the block dependencies.

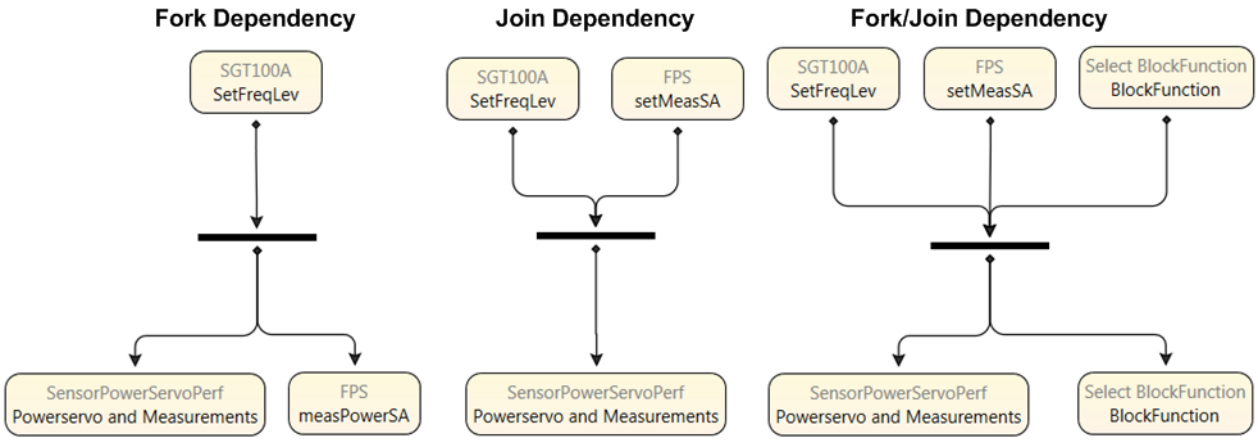


Figure 4-16: Fork/Join element in a test procedure

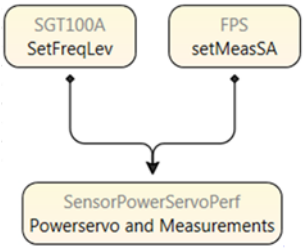


Figure 4-17: Alternative join dependency

- Fork dependency: After the block function at the fork base has been executed, the block functions at the fork ends are executed in parallel.
- Join dependency: The block function at the joining end comes into action after the selected functions of all blocks at the originating ends have been executed.
  The execution of the block at the join end is inhibited even if the execution of only one originating block does not come to an end. This behavior can cause an unwanted stop of the test run and can be avoided with an appropriate condition for the block, see Chapter 4.5.3.1, "Execution Conditions", on page 39.
- Combined join and fork dependency with Fork/Join element: This structure has the same behavior as if the arrow lines have been drawn directly from the originating blocks to the terminating blocks. In case of three incoming block dependencies (from blocks 1, 2, 3) and two outgoing dependencies (to blocks A and B): Block A comes into action after blocks 1, 2, 3 have been processed. Block B behaves in the same way. Consequently, block A and block B are executed in parallel.

Control statements: If, Or, End

The course of block functions executed during test execution can be controlled with "If" and "Or" statements. In this way, block functions can be repeatedly executed in one test step, for example in combination with loops. The end of an assembly of block functions can be stated by the "End" element. The following figure shows by example how these elements are used.

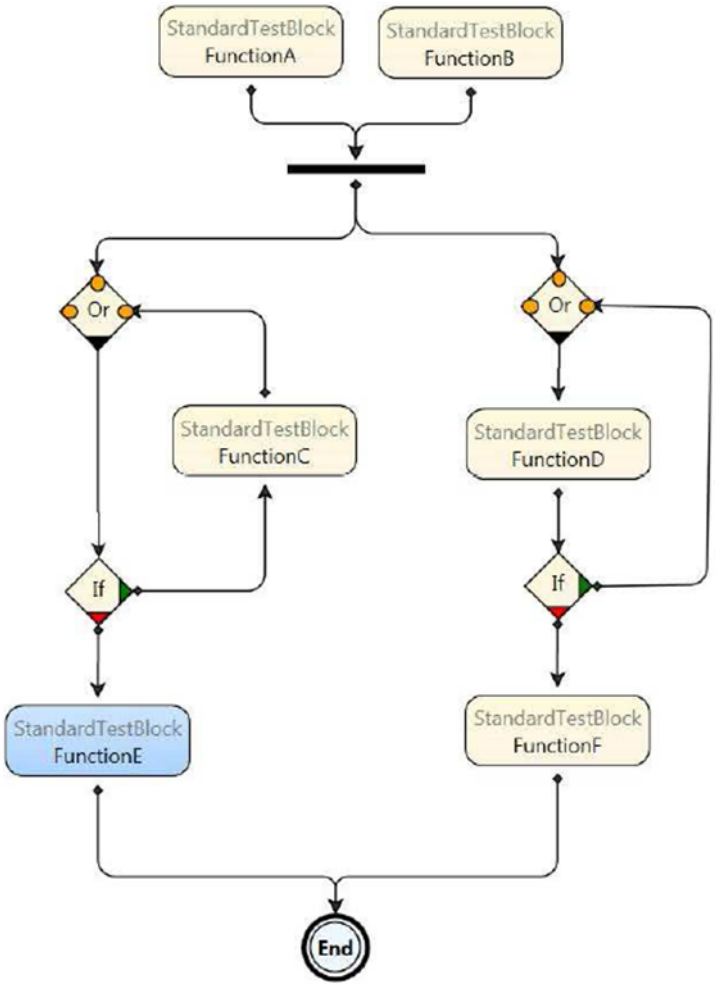


Figure 4-18: Control statements in a test procedure

The "End" statement is only required if "If" or "Or" statements are used in the block flow chart (to avoid that the test execution phase ends when one branch execution is completed).

The "End" statement can be used in two ways:

- Several branches end in one single "End" statement: The execution phase ends as soon as the block functions on all input connections to the "End" statement are completed.

- The test procedure branches terminate in several different "End" statements: The execution phase ends when the first "End" statement is reached. Already started block function are completed, but no new block functions are started in that execution phase. The following execution phases are started as usual.

For more information, see the QuickStep Training Manual.

4.5.5 Parallel and Synchronized Test Execution

4.5.5.1 Parallel Execution of Block Functions

A parallel execution of block functions is often useful to shorten the overall test time. Therefore, the block functions are arranged in parallel in the "Test Procedure".

If functions of the same block shall be executed in parallel, a second condition is required for real parallel execution: The block functions must belong to own block instances. The mechanism is illustrated by example with two block functions:

Case A: Only quasi-parallel execution

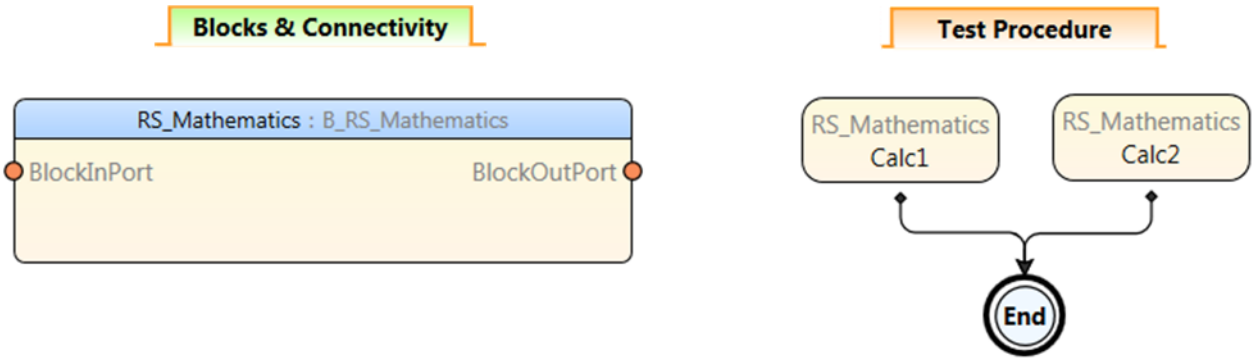


Figure 4-19: Quasi-parallel block execution

The block for the used block functions is created once in the "Blocks & Connectivity" section resulting in just one block instance. One block instance defines one thread and both block functions are executed in this thread. The block functions send communication messages in parallel according to their arrangement in the test procedure. Since there is only one block instance, the messages get into the incoming queue of the block instance and are processed sequentially.

Case B: Parallel execution

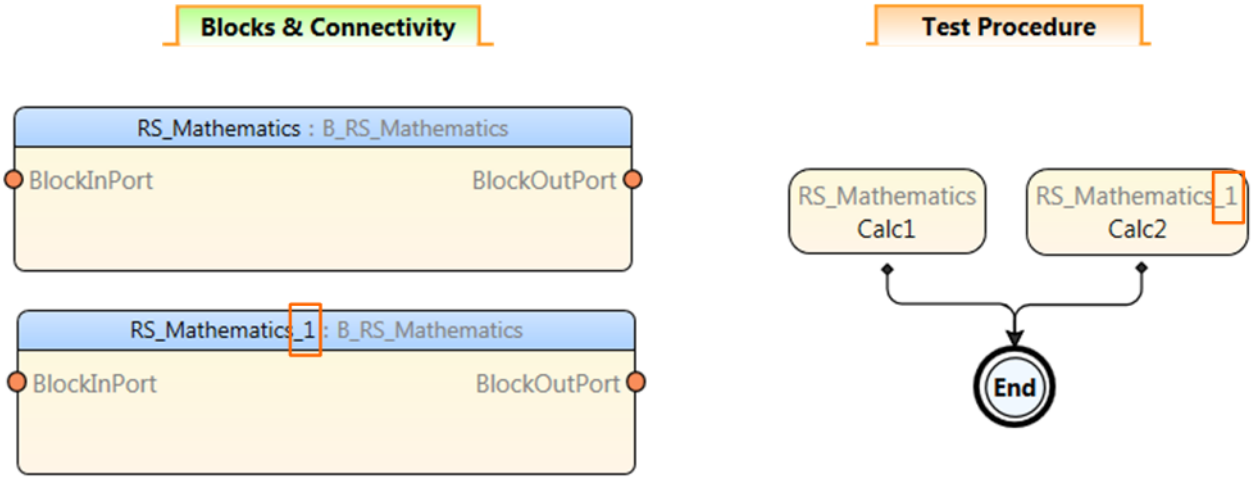


Figure 4-20: Parallel block execution

The block is created twice in the "Blocks & Connectivity" section resulting in two instances of the block and two threads for block function execution. The block functions are executed in separate threads, so they are executed in parallel.

4.5.5.2 Synchronization during Test Execution

The figure shows the synchronization points during execution of a test including the test phases.

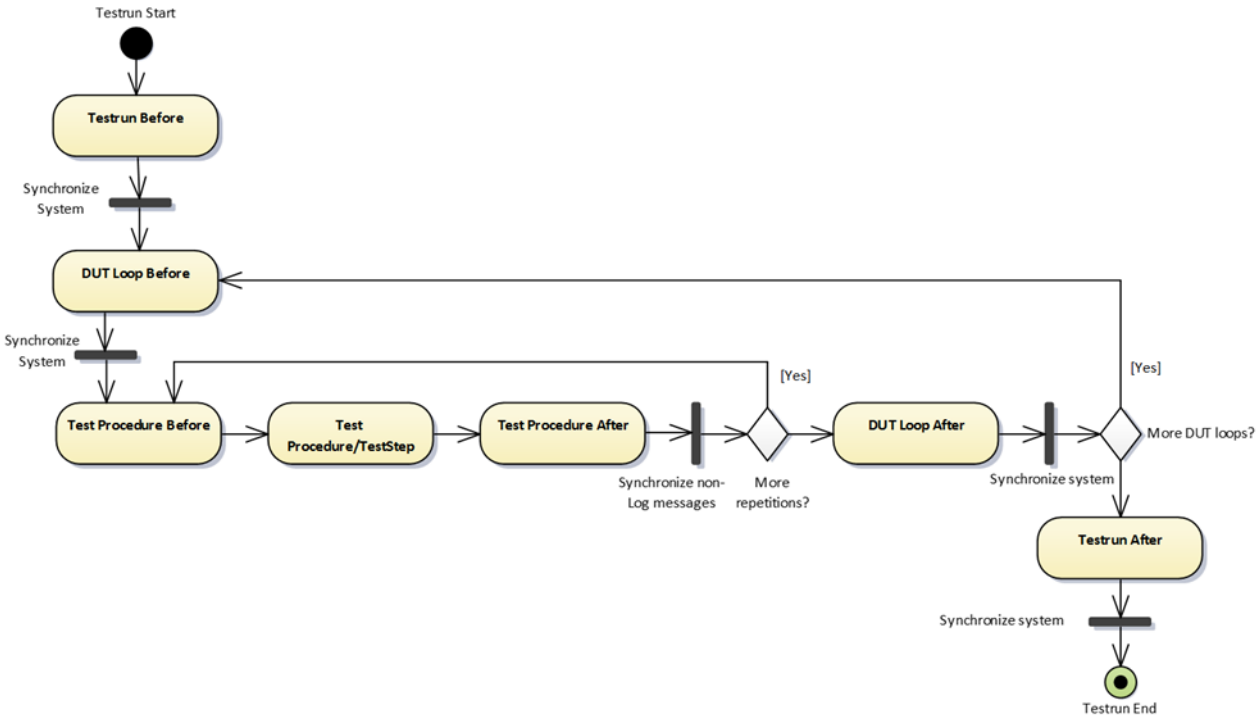


Figure 4-21: Synchronization

Synchronization takes place at the Join elements (black bars). “Synchronize non-Log messages” waits for all block functions to be completed. For example, a block function calls another block function asynchronously with the called block function not contained in the block function flow chart of the test procedure (implicit call). If the block functions were not synchronized, block functions of different test steps would run in parallel, and this behavior is not intended.

Log messages do not influence the functionality. Therefore, they can be synchronized after all repetitions are done. The log messages are synchronized with “Synchronize System”. The logging information is separated from the test procedure execution to guarantee high performance. This leads to a performance optimized processing of the log messages which uses idle time during test execution.

4.6 System Configuration

The system configurator models the physical test setup in terms of devices, RF connections and paths (groups of contiguous RF connections and attenuator components). Frequency-dependent attenuations can be defined. Different test benches for a test project are modeled with different system configurations.

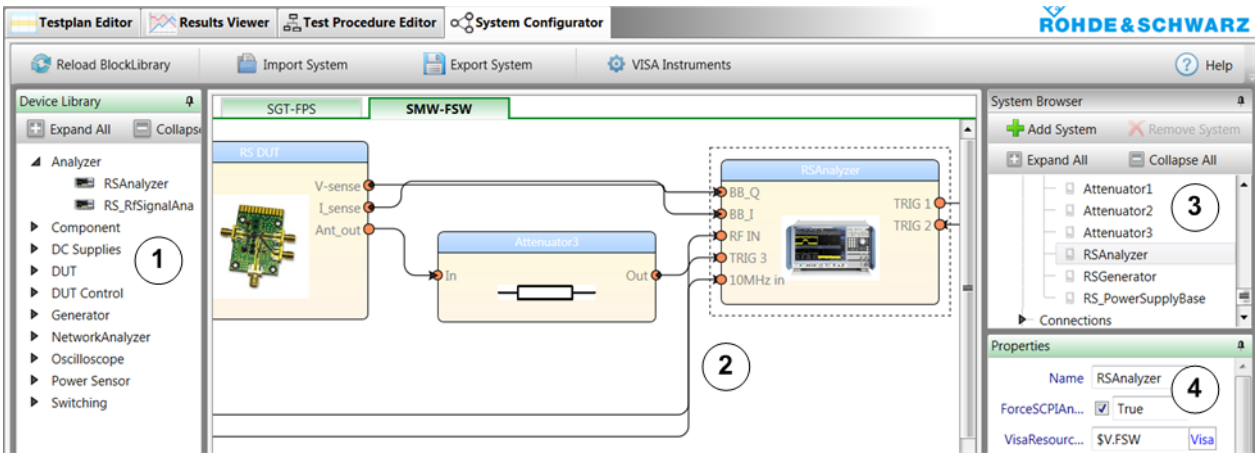


Figure 4-22: System configurator

1 = Drag a symbol into the main area
2 = Connect the symbols
3 = Select a symbol or connection
4 = Edit the properties (parameters) of the selected device/connection

For using a system configuration in a test, test plan parameters are mapped to parameters of that system configuration. Consequently, the parameter values from the system configuration are used for the test plan execution. Actually, when building a new system configuration, the system and the symbols for the test instruments have no associated parameters at first; the "Properties" view is empty. The parameters for a system configuration are created with the "Mapping Table Editor" accessed via the "Testplan Editor", see "Creating and using additional system configuration parameters" on page 52.

System configurations and mapping keep test procedures free from the test setup details and to switch easily between different test benches.

Using the "System Configurator" is mandatory if path attenuations are used. For more details about RF attenuations, see Chapter 4.8, "Handling of RF Attenuations", on page 53. Otherwise the usage is optional.

Use cases

- VISA resources of the test instruments (closely related to the test setup) are set in the system configuration and used in the test procedure after appropriate mapping.
- Attenuations for RF connections, additional RF components (for example splitters) and RF paths are set in the system configuration and used in the test procedure after appropriate mapping. It is often more convenient to handle path attenuations than the attenuations of all the single path components.

- Several test benches are available, for example the test setup is realized with different types of test instruments. All system-specific settings (attenuations, VISA recourses, …) are centrally managed within the system configurator. Handling:
  - The different test benches are described by different system configurations.
  - The test procedure contains general block functions suitable for different devices.
  - Parameters of the general block functions are mapped to the desired system configuration and its elements. Selecting one of the system configurations automatically selects all system-specific settings.
- The test setup contains more than one instrument of one type (for example two R&S NGMOs): Set instrument parameters according to the instrument's position in the test setup, i.e. in the system configuration.

Element groups

In the "System Browser", the elements of the main pane are collected in the following groups:

- "Devices": Analyzers, generators, splitters, attenuators, ...
- "Connections": Described by originating and terminating end, for example "Ant_out -> In".
- "Paths": Output (out direction at DUT connector) and input paths. See the figure below.

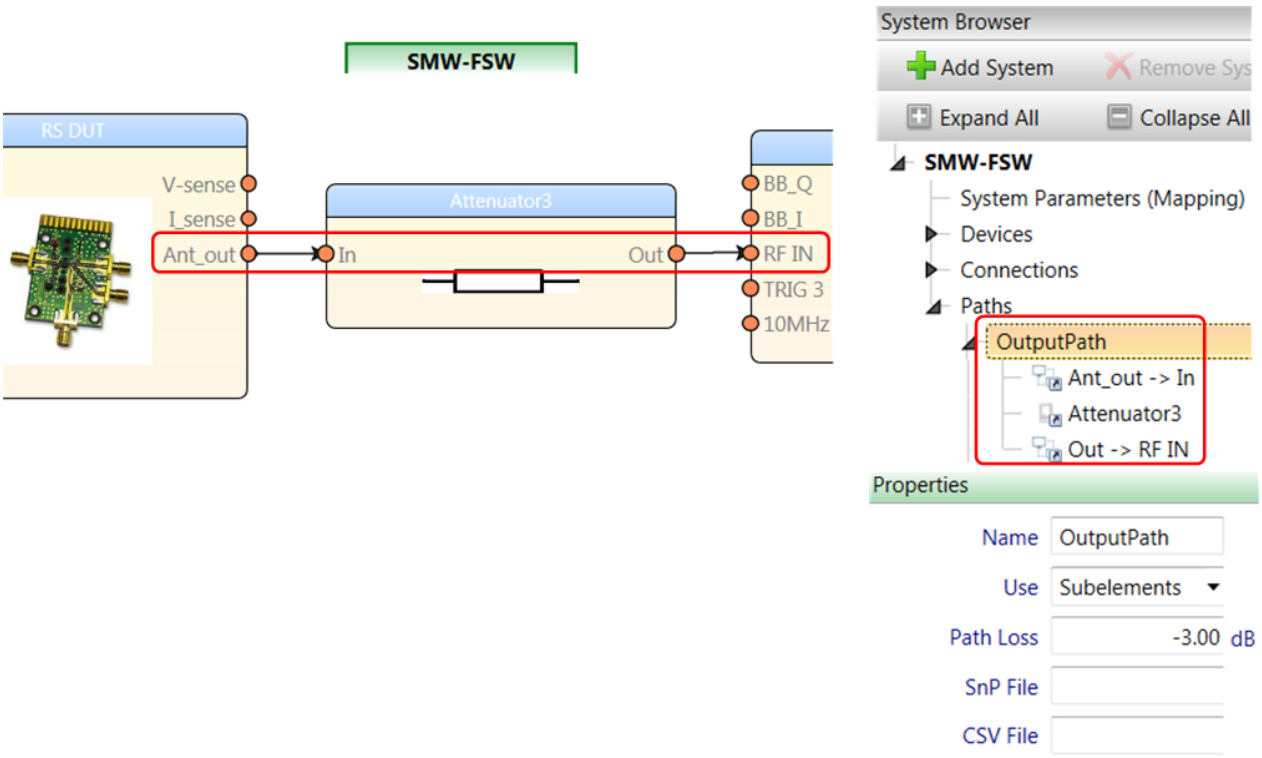


Figure 4-23: Structure of a system configuration path

System definition file

A system configuration defined by the "System Configurator" is embedded in the `*.tpl` test plan, but it can separately be exported and imported as `*.sdf` system definition file. In this way, sharing of system configurations is supported.

4.7 Setting Parameter Values by References

QuickStep provides referencing functionality: Parameter A can get the value of parameter B by reference, meaning that the value entry of parameter A is a reference to parameter B.

Use cases:

- Several block functions require the same parameter (for example “Frequency” at generator and analyzer block functions): It would be inconvenient to set the parameter values at each block function. Instead, a common parameter (for example a test procedure parameter) is defined and the analog parameters of the block functions get their values by referencing to this common parameter.

- During test run, a result of a test step or block function is used as input for a following test step or block function, for example:
  - The input parameter for the current test step gets its value by reference to the result variable of another test step.
  - An out parameter value of an executed block function determines if a following block function is carried out or skipped (dynamic execution condition).
- Parameters which are closely related to the test setup and are used in the test plan: The values of such parameters are best set in a system configuration. The test plan parameters get their values by referencing to system configuration parameters. Note that certain system configuration parameters have to be created via "Mapping Table" within the "Testplan Editor" before they are available for referencing, see "Creating and using additional system configuration parameters" on page 52. See also "Best practice: Referencing to several system configurations" on page 53.
- VISA resource strings are required for several test instruments: The VISA resource strings are collected in one table. The resource string parameter of an instrument block function gets the required string by reference to the table.

The reference to a target parameter is realized as reference string set in the input field of the referencing parameter. The reference string is composed of a starting *$* followed by the area indicator (describing where the referenced parameter is located) and the target parameter's Id. See the table below for details and additional options.

Referencing via the "Set Reference" dialog is more convenient. This dialog is available in various contexts, for example for parameters in the "Test Step Parameters" tab within the "Testplan Editor". The "Set Reference" dialog is opened when a parameter input field is right-clicked and "Set Reference" is selected. See "Set Reference dialog" on page 209 for details.

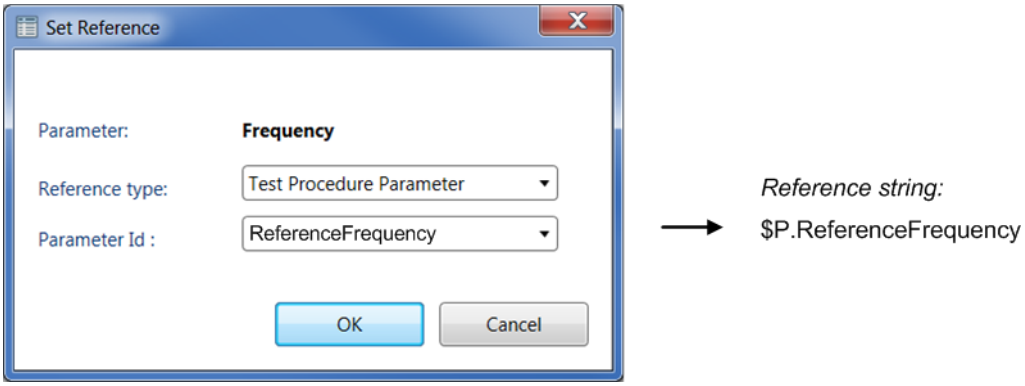


Figure 4-24: Set Reference dialog

Table 4-1: Reference strings

Reference area ("reference to")	Prefix	Reference string, example	Description
Test project	*$T.*	*$T.Band*	Reference to the test project parameter with Id "Band"
Test procedure	*$P.*	*$P.Frequency*	Reference to the test procedure parameter with Id "Frequency"
Result	*$R.*	Full version: *$R.<Block><BlockFunction><ParameterName><RepNo><TestStep><LoopId>*	
		$R.<RSAnalyzer><Measure EVM><EVM><this><last><this>	Reference to the EVM result in current repetition and loop, last test step. The EVM result belongs to the RSAnalyzer block and the Measure EVM block function
		Standard version: *$R.<Pin><3><7>*	Reference to the In Power result in repetition 3 and test step 7
		Short version: *$R.<Pin>*	Reference to the current In Power result
System configuration	*$M.*	*$M.MappingParameter*	Reference to the resource mapping parameter of the current system configuration
	$V.	*$V.VisaAlias*	Reference to a VISA resource string configured in the “VISA Instruments” section in the System Configurator
Test project	*$G.*	*$G.GlobalVariable*	Test project variable, used in combination with an out parameter of a block function

Note the difference between test project parameter ($T) and test project variable ($G): The value of a test project parameter is fixed during a test run while the test project variable's value can change.

How to: See Chapter 6.4, "Setting Values by Reference", on page 97

How to: See Chapter 6.5, "Using a Block Function Result as Input for Another Block Function", on page 101

Note that in the testplan table all values which come from a reference are displayed in blue font.

For more details (for example concatenating references) and examples, see the QuickStep User Training manual.

Creating and using additional system configuration parameters

Additional system configuration parameters (visible in the "System Configurator") can be created and mapped to devices. This action is done with the "Parameter Mapping Table" in the "Testplan Editor", not in the "System Configurator".

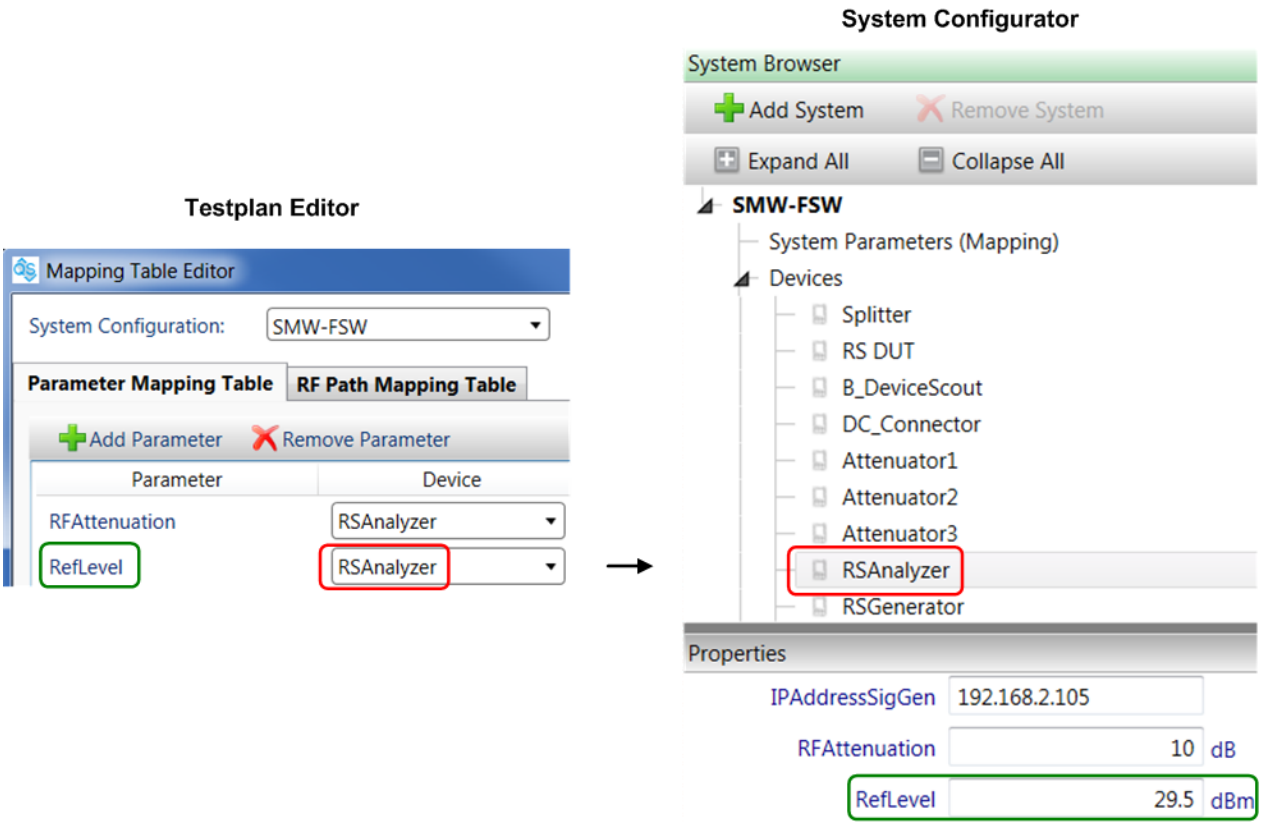


Figure 4-25: Mapping of parameters to system configuration devices

Such additional system configuration parameters are useful for providing values of test plan or test procedure parameters by references. The mapping mechanism allows you to use the same system configuration parameter for different devices and to change the device without having to change references to the parameter.

See Chapter 6.9, "Using System Configuration Parameters", on page 107 for further information. For information about using system configuration paths, see Chapter 4.8.2, "Compensation of RF Attenuations via System Configuration Paths", on page 54.

Best practice: Referencing to several system configurations

Use case: A test plan shall be executed with several test setups. The test setups consist of the same functional components (analyzer, generator, ...) but are realized with different device types (analyzer type A, analyzer type B, ...). The device types of the same functional component are controlled by the same parameters but with different values.

Solution: A test plan parameter is mapped to the same device for the different system configurations. So, if the active system configuration is changed, the referencing parameter gets its value from the new system configuration.

4.8 Handling of RF Attenuations

Connections (cables, connectors) between DUT and test instruments as well as other devices (e.g. splitters, attenuators) in the RF signal paths produce signal attenuations. These attenuations are compensated in order to avoid inaccurate results. The compensation makes sure that, for example, the test instruments indicate exactly that power values which are present at the DUT's ports. In a typical application, the attenuations for an output path (signal from the DUT) and an input path (signal to the DUT) are compensated separately.

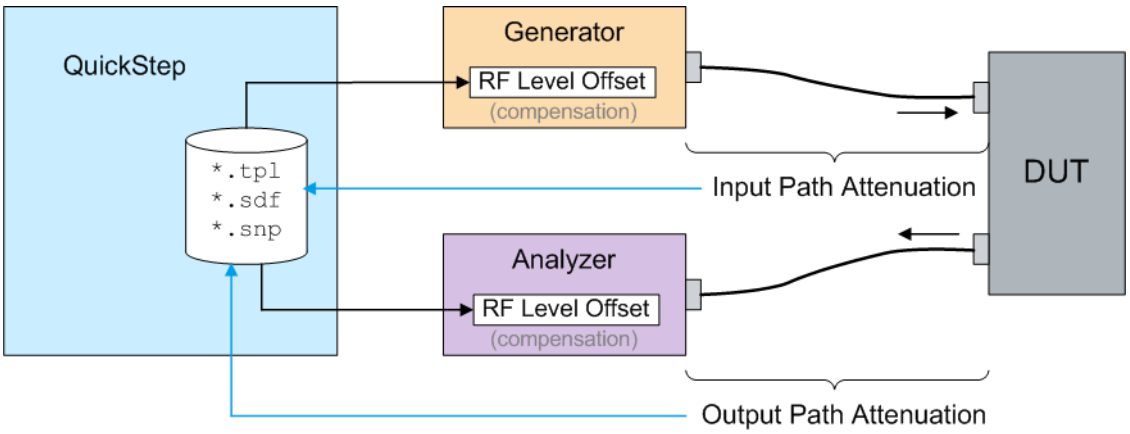


Figure 4-26: Compensating in and out attenuations

4.8.1 General Methods

Frequency-independent compensation: The most simple way of compensation uses just one fixed in attenuation value and one fixed out attenuation value for all attenuation contributions in an RF path. Some test procedures provide RF Level Offset parameters which can be used for compensating the attenuations. For a better transparency, it is recommended to use path compensation as described below.

Frequency-dependent compensation: This method requires attenuation tables, each row containing the attenuation values for one frequency. The standard files for this purpose have the `*.s3p` or `*.s4p` format (SnP files, Touchstone format) and are typically generated using network analyzers. Alternatively, `*.csv` files can be used which allow to store frequency-dependent information in a simpler way.

All loss files to be used by a specific QuickStep project have to be placed in the project sub folder:
`C:\...\ProjectName\ConfigurationFiles\LossFiles`

4.8.2 Compensation of RF Attenuations via System Configuration Paths

QuickStep handles attenuations with path objects in the System Configurator and with path parameters of test procedures: The path objects of the System Configurator define the attenuations, and a system configuration path is selected as value for a test procedure's path parameter ("RF Path Mapping"). In this way, the

attenuations defined in the System Configurator are applied by the test procedure.

Paths in the System Configurator

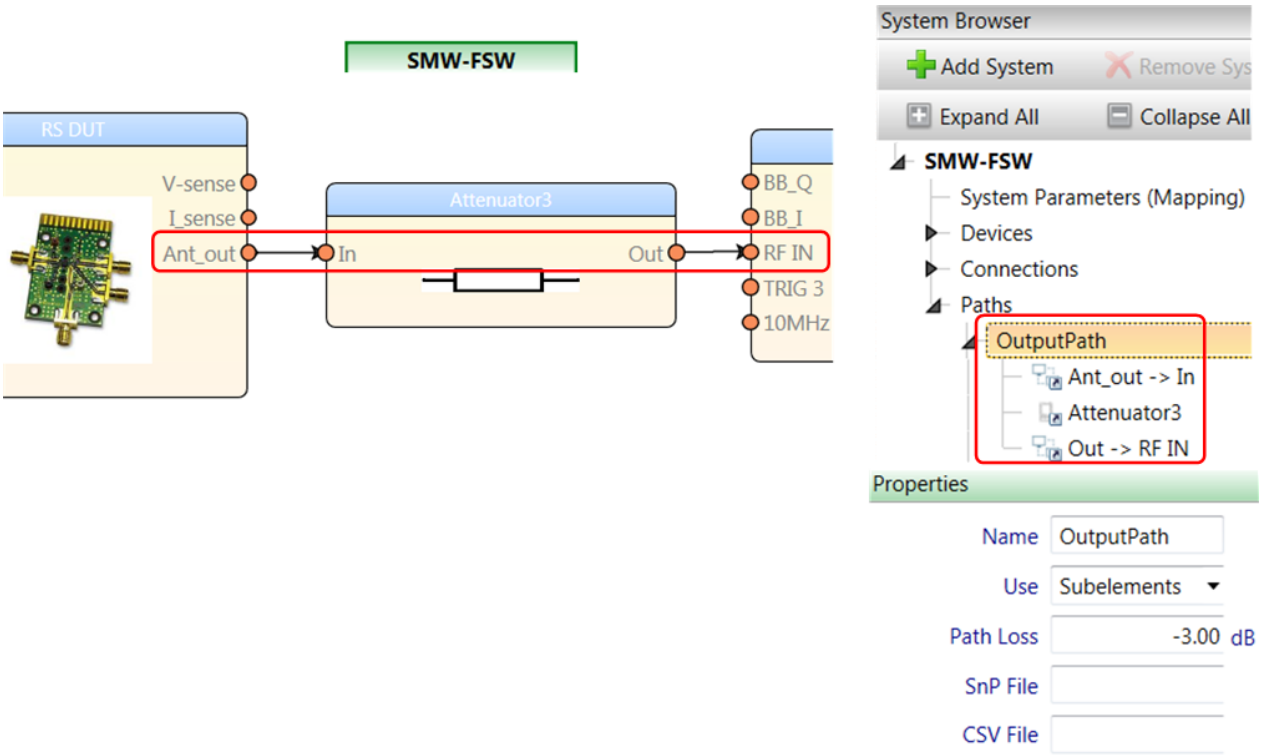


Figure 4-27: Paths in the System Configurator

Multiple input and output paths can be defined in the System Configurator. Each path collects RF elements (whose individual attenuations are separately assigned in the "Devices" and "Connections" nodes in the System Browser). The "Use" parameter of the path "Properties" defines how the attenuation of the path is calculated. "Use" has the following values:

- "Subelements": The path attenuation is calculated as sum of the attenuations of the individual path elements.
- "Path Loss": The attenuation is a fixed, frequency-independent value as defined in the "Path Loss" parameter. Note that path losses have to be entered as negative numbers.
- "SnP File": The path attenuation is taken from the `*.snp` file as defined in the "SnP File" parameter (frequency-dependent compensation).
- "CSV File": The path attenuation is taken from the `*.csv` file as defined in the "CSV File" parameter (frequency-dependent compensation).

RF Path mapping

Block functions can have parameters of type RF_Path (defined in the Block Development Tool). Parameters of this type can directly be filled with a number of type double (negative value for a loss (attenuation), positive value for a gain) or a name can be entered. If a name is entered, this name has to be linked to a path in the System Configuration using the "RF Path Mapping Table".

In the figure, the Output Path parameter is of type RF_Path and has the name value SigAnOutputPath. This name appears in the left column of the RF Path Mapping table. Here, SigAnOutputPath is mapped to OutputPath of the System Configuration. In the end, the Output Path parameter gets the value of Output-Path as set in the System Configuration during test execution.

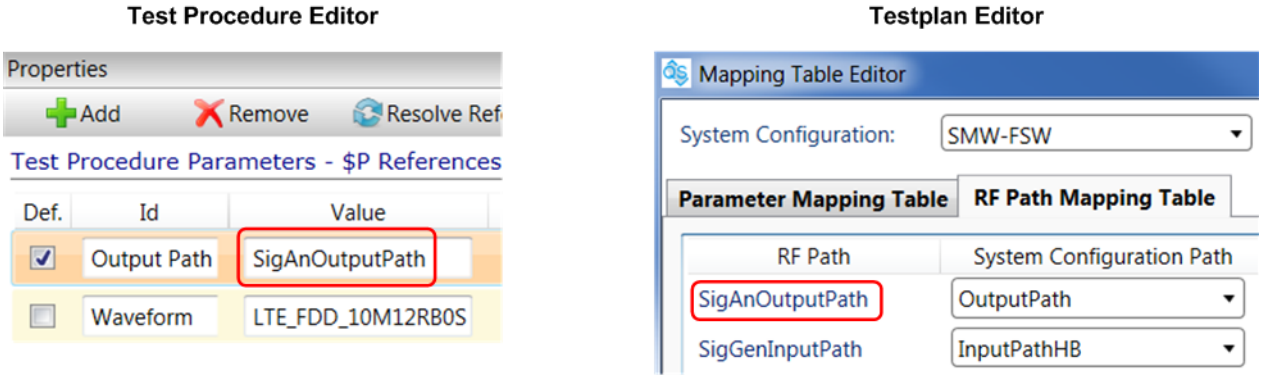


Figure 4-28: RF Path Mapping

This mapping approach decouples the test plan from the system configuration and therefore maximizes the flexibility.

4.9 Limit Handling

Limits are used for checking if results are within desired value ranges and for triggering predefined actions if this is not the case. They basically define maximum or minimum values for a measurement result variable. The limits can be set specifically for each test step or for higher test plan entities like a complete test sequence. If a measurement result is outside its limits, a failure is reported.

Limit setting is realized in two stages: First, limits are defined as arrays which carry maximum and minimum values (and optionally additional information). Second, a result variable is connected with such a limit, thereby getting the limit's values. The limits are defined in an Excel sheet which is imported in QuickStep.

Limit table, setting result limits

Limits for a test are defined in an Excel limit table. The excel file contains two sheets. In the “Limits” sheet, each table row begins with the name of a limit and is followed by one or more minimum, maximum and binning values. In the “Binning" sheet, softbins and hardbins are defined. Each softbin is linked to a specific action and a specific hardbin.

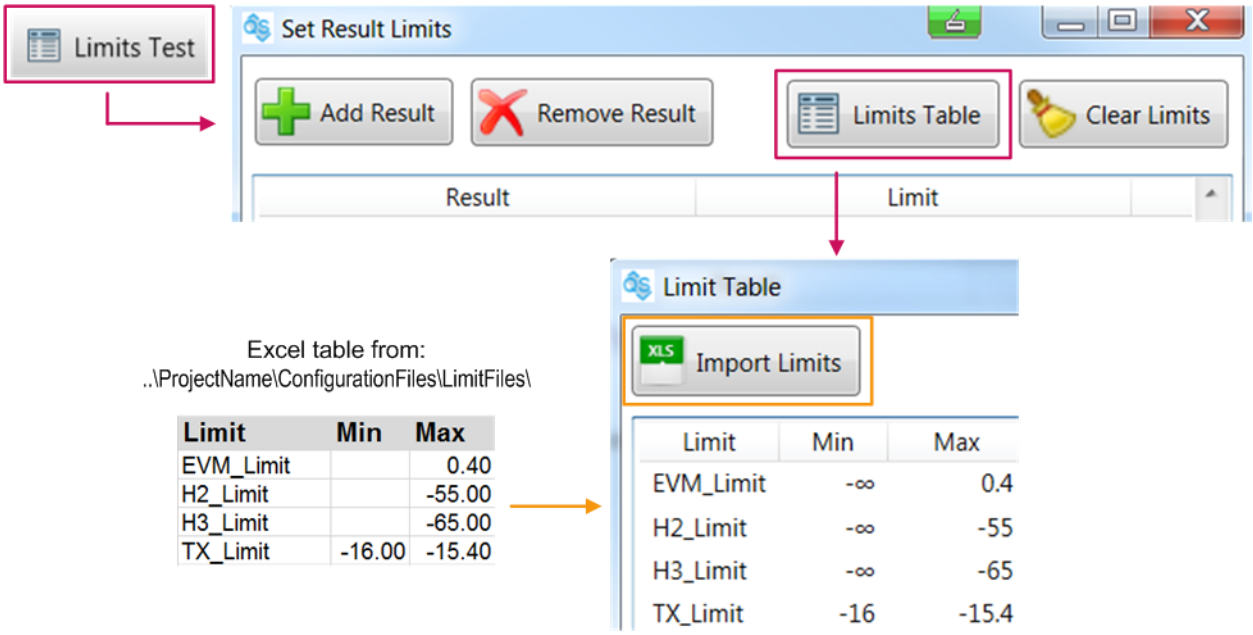


Figure 4-29: Import of a limit table

To apply limits for a result variable, first the Excel limit table has to be stored under the directory `C:\...\ProjectName\ConfigurationFiles\LimitFiles\` and then to be imported from there into QuickStep. The import is requested via the "Limits Test" button in the toolbar of the "Test Plan Editor" and then the "Import Limits" button in the "Limit Table" dialog. The dialog additionally shows the current limit table according to the imported Excel table.

The result variables of QuickStep are manually connected with imported specific limits in the table of the "Set Result Limits" dialog. So, QuickStep does not show the limit values for a result variable but the connected limit name.

Result variables and connected limits – if available – are also shown in the "Test Step Limits" tab on the right side of the "Testplan Editor".

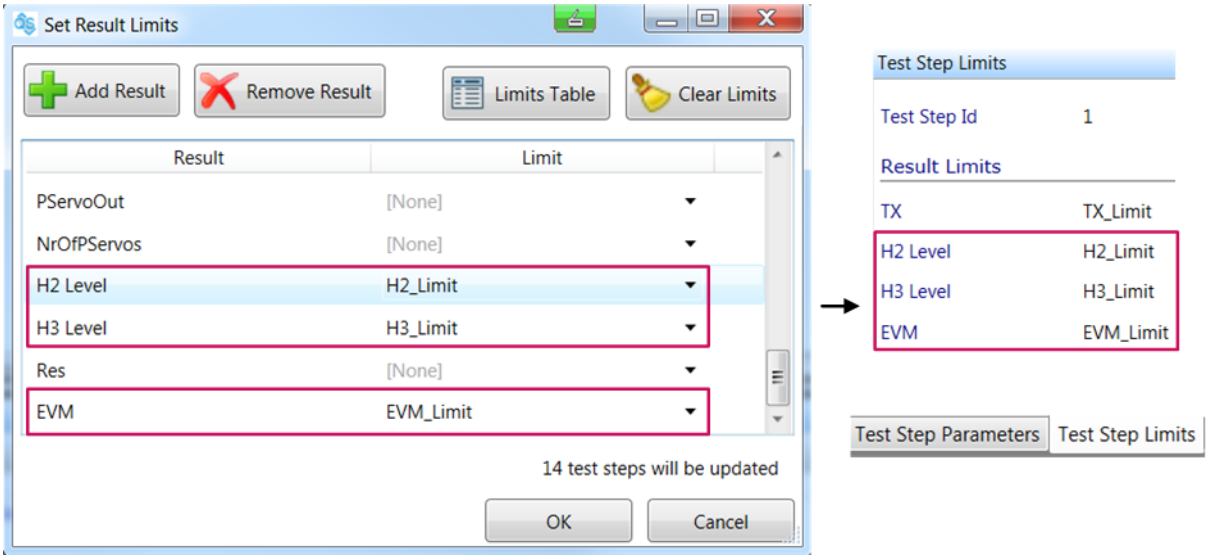


Figure 4-30: Setting result limits

Which result variables are available and displayed in the "Set Result Limits" table is determined by the used functional blocks in the test procedure. QuickStep finds the result variables mainly by searching for `Send…LogResult…(…)` commands in the code. The names of the result variables must be unique and each name must consist of one contiguous string to avoid any conflicts.

The "Set Result Limits" table is empty if no test step or sequence (or higher test plan entity) has been selected or if no Excel limit table has been imported.

Validity area for limits

Limits set via "Set Limits" from the toolbar are applied to one or several test steps which have been activated before by clicking one or more test step rows. For applying limits to a complete sequence, group or the test plan itself, a "Set Limits" context menu is available for sequences, groups and test plan in the "Test Browser".

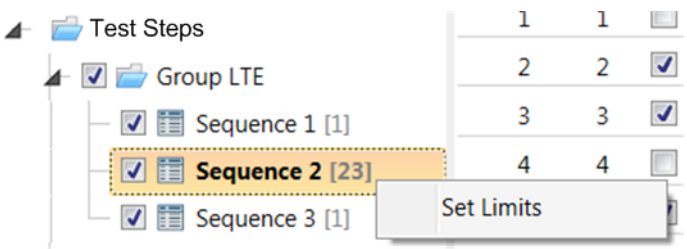


Figure 4-31: Setting limits for a complete sequence via Test Browser

The limit checks and the behavior in case of failures can be defined in the TPR options.

Binning

QuickStep supports binning which is an extended mode of limit handling: Several limit sets (min and max) can be defined for a result parameter determining the severity of a failure. The test results are grouped in several containers ("bins") according to the failure severity.

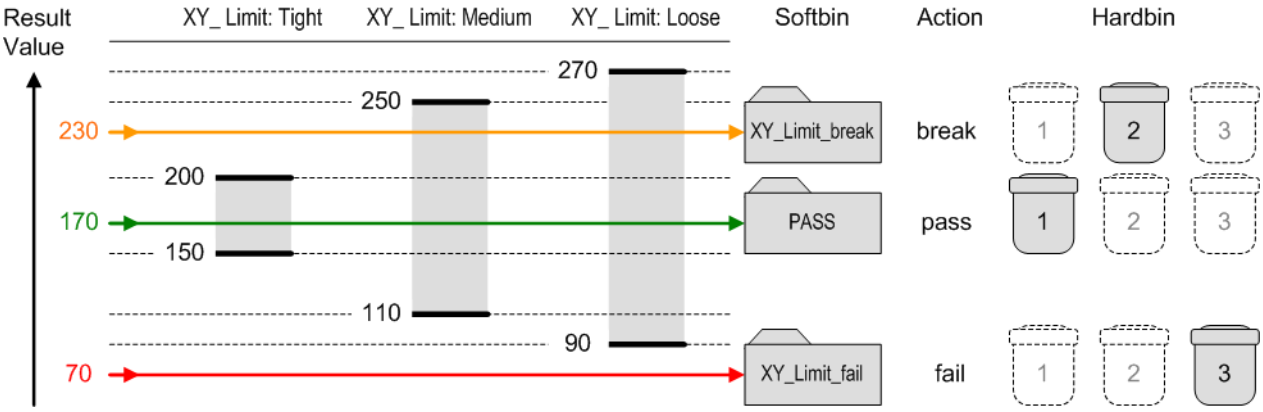


Figure 4-32: Principle of binning

Characteristics (according to the figure):

- Each result value for a result variable is submitted to a sequence of limit checks. A limit check is passed if the result value is within the associated limit set (usually defined by a maximum and minimum value). If not, the result value fails that limit check. The limit gates (XY_Limits in the figure) are specific for the result variable (XY).
- The first limit gate has to be tighter than the second limit gate which has to be tighter than the third limit gate. So, if a measurement value passes one gate, it also passes the following gate.
- **Softbins:** Each limit set of the result variable contains a softbin. The softbin which a measurement value is assigned to is determined by the last limit check failure if there is any. The softbins are specific for the result variable (as the limit gates).
  - A result value which passes the tightest limit gate is assigned to the first softbin which is named "PASS".
  - If a result value fails only on the tightest limit gate, it is assigned to the second softbin "XY_Limit_break" for that result variable.

  - If a result value fails on the first and second limit gate but passes the third one (if available), it is assigned to the third softbin for that result.
  - If a result fails on all limit gates, it is assigned to the last softbin for that result, here "XY_Limit_fail".
- Each softbin is related to an action:
  - Pass: The test execution is continued.
  - Break: The active test step is finalized, then the test execution halts with a pop-up window. Options are provided to select how to continue.
  - Fail: The active test step is finalized, then the test execution is stopped.
- **Hardbins:** Hardbins allow a second level of grouping of several softbins into one hardbin. In a production environment, a handler can be used to sort the DUTs after testing in several physical trays. The handler could be controlled using the hardbin number. In the example, hardbin "1" collects the result values without failure, hardbin "2" collects the failures with associated "break" action, hardbin "3" collects the failures leading to the "failed" action.

The softbins and hardbins are defined in the second sheet of the Excel limit table to be imported into QuickStep.

Limit	Min	Max	Softbin	Min	Max	Softbin	Min	Max	Softbin
EVM_Limit		0.40	EVM_Limit_break		0.42	EVM_Limit_ext_break		0.44	EVM_Limit_fail
H2_Limit		-55.00	H2_Limit_break		-52.00	H2_Limit_ext_break		-50.00	H2_Limit_fail
H3_Limit		-65.00	H3_Limit_break		-61.00	H3_Limit_ext_break		-60.00	H3_Limit_fail
TX_Limit	-16.00	-15.40	TX_Limit_break	-16.30	-15.10	TX_Limit_ext_break	-16.60	-15.00	TX_Limit_fail

Limits | Binning

Figure 4-33: Definition of softbins and limits

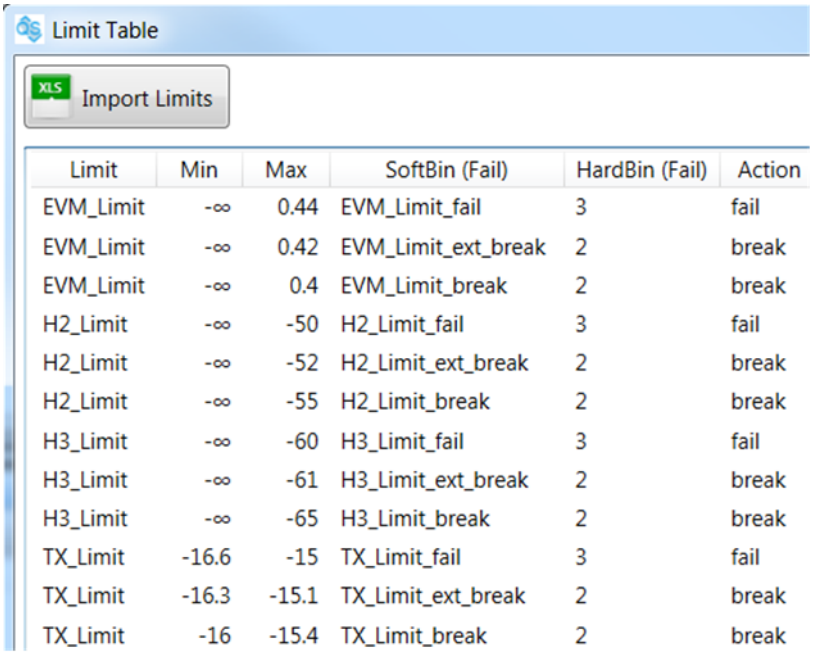
Softbin	Name	Action	Hardbin	Description
1	PASS	pass	1	Test passed
2	EVM_Limit_break	break	2	Test halted but can be restarted
3	EVM_Limit_ext_break	break	2	Test halted but can be restarted
4	EVM_Limit_fail	fail	3	Test failed
5	H2_Limit_break	break	2	Test halted but can be restarted
6	H2_Limit_ext_break	break	2	Test halted but can be restarted
7	H2_Limit_fail	fail	3	Test failed
8	H3_Limit_break	break	2	Test halted but can be restarted
9	H3_Limit_ext_break	break	2	Test halted but can be restarted
10	H3_Limit_fail	fail	3	Test failed
11	TX_Limit_break	break	2	Test halted but can be restarted
12	TX_Limit_ext_break	break	2	Test halted but can be restarted
13	TX_Limit_fail	fail	3	Test failed

Limits | Binning

Figure 4-34: Binning table

QuickStep executes the actions break and fail listed in the Action column in the Excel Binning tab if "Continue on Fail" is deactivated in the "TPR Options" of the test plan.

After import of the limit table, the softbins and hardbins are shown together with the limits.



Limit	Min	Max	SoftBin (Fail)	HardBin (Fail)	Action
EVM_Limit	-∞	0.44	EVM_Limit_fail	3	fail
EVM_Limit	-∞	0.42	EVM_Limit_ext_break	2	break
EVM_Limit	-∞	0.4	EVM_Limit_break	2	break
H2_Limit	-∞	-50	H2_Limit_fail	3	fail
H2_Limit	-∞	-52	H2_Limit_ext_break	2	break
H2_Limit	-∞	-55	H2_Limit_break	2	break
H3_Limit	-∞	-60	H3_Limit_fail	3	fail
H3_Limit	-∞	-61	H3_Limit_ext_break	2	break
H3_Limit	-∞	-65	H3_Limit_break	2	break
TX_Limit	-16.6	-15	TX_Limit_fail	3	fail
TX_Limit	-16.3	-15.1	TX_Limit_ext_break	2	break
TX_Limit	-16	-15.4	TX_Limit_break	2	break

Figure 4-35: Imported limits with SoftBins and HardBins

Failure report

The failures due to limit violations are reported in the execution log and in the execution protocol available via the "Results" tab. A failure report includes the name of the assigned softbin.

At the end of the execution protocol, a failure report of that test step is repeated where the first limit violation occurred. It includes the softbin assigned to the first result variable which caused a limit failure in that test step.

Results Table: **\ExampleEtDut\ExecutionProtocol_0.txt**

Block	Log
QuickStepEngine	**********[LF]
QuickStepEngine	********** Test failed! **********[LF]
QuickStepEngine	**********[LF]
QuickStepEngine	Repetition '1' Teststep '2' failed at Softbin 'H3_Limit_break' with value: 64.32

Figure 4-36: Failure report

4.10 DUT Handling

Typically, it is more efficient to test several DUTs in a row within one test than with separate tests: The initial test execution phase ("Testrun Before" typically including initializations of the test instruments) has to be executed only once. For handling several DUTs within one test, a DUT loop mechanism is available which relies on the execution phases "DUT Loop Before" and "DUT Loop After", see the following figure.

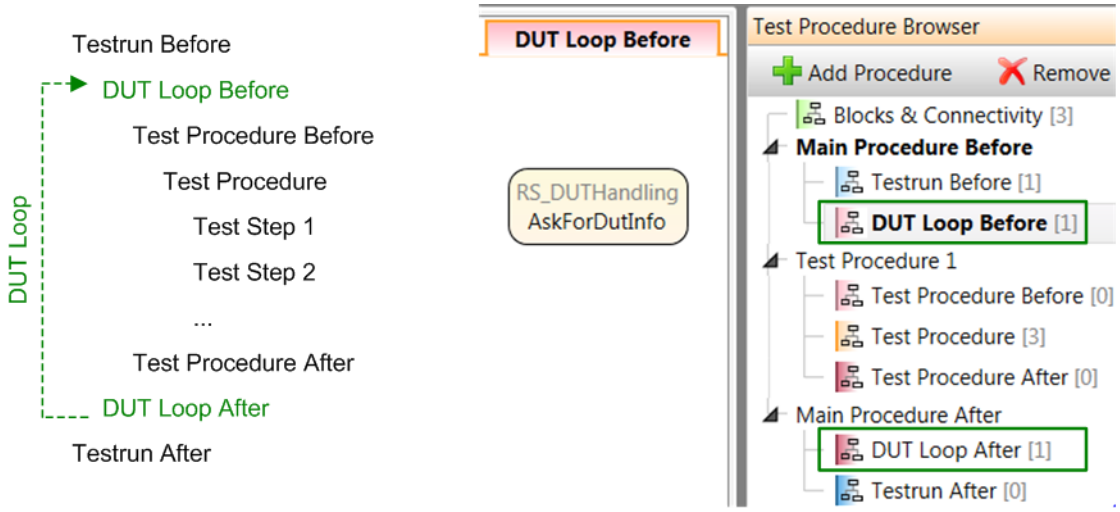


Figure 4-37: Test execution phases for a DUT loop

The "DUT Loop Before" phase is used for specifying the DUT. The "DUT Loop After" phase can carry any desired block functions. Looping back from the "DUT Loop After" phase to the "DUT Loop Before" phase is done automatically based on user input during test run. So, the number of repetitions is dynamic. A special block, "DUT_Handling", is provided for entering DUT information, configuring

information elements to be logged and ending the DUT loop. The functionality is contained in the block functions "AskForDutInfo" and "LogDutInfo".

For testing several DUTs in a row via the DUT loop mechanism, the test execution has to be started by clicking “Continuous Run” from the "Testplan Editor" toolbar. “Single Run” ends the test execution after the first DUT test is completed.

AskForDutInfo block function

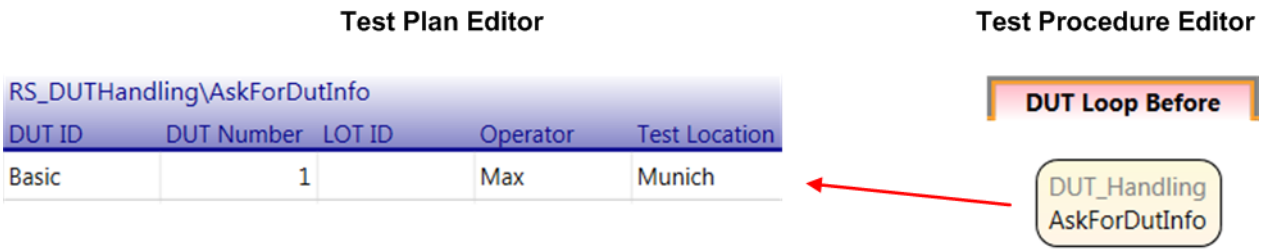


Figure 4-38: AskForDutInfo in DUT Loop Before

The "AskForDutInfo" block function is placed in the "DUT Loop Before" phase. When it is called (that is at begin of each DUT loop repetition), the "DUT Information" dialog is displayed for entering information about the DUT and the test situation. In this way, test results can be related to the used DUTs.

When clicking the "Stop Testing" button in the "DUT Information" dialog during test run, the test execution is stopped immediately (the current DUT cycle is not executed anymore).

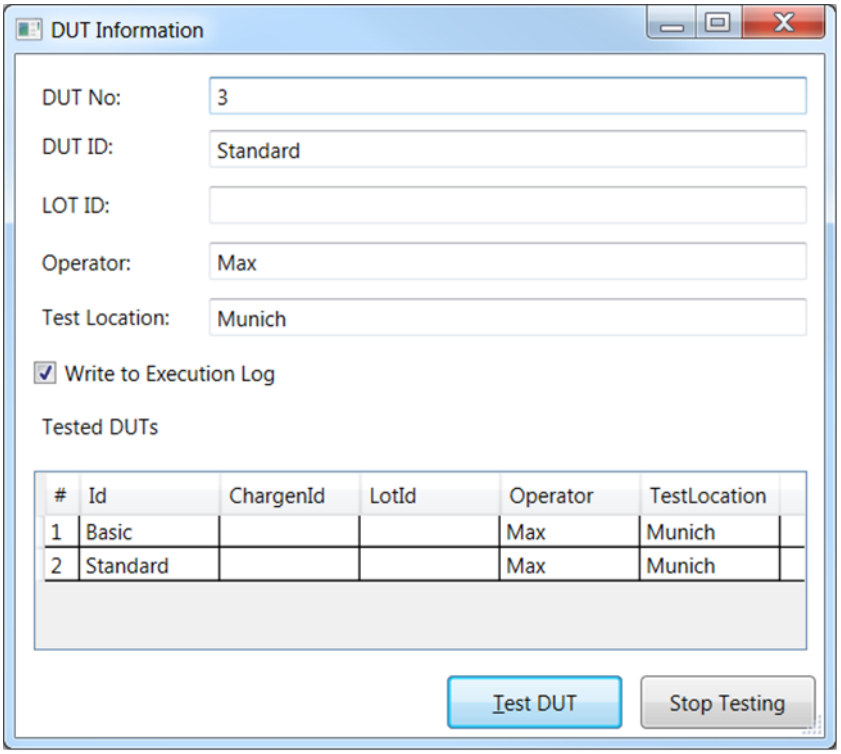


Figure 4-39: DUT information dialog

LogDutInfo function

The "LogDutInformation" block function is placed in the "Test Procedure" phase in the "Test Procedure Editor". It allows to select which items are stored in the `TestStepsResults.log` file.

Characteristics:

- The parameters of the function reflect the DUT information elements listed in the "DUT Information" dialog (see the "AskForDutInfo" block function). If a parameter is activated via check box, the information element appears as column header in the results table of the `TestStepsResults.log` file. The values of the selected DUT information elements are given for each test step.
- For each DUT, a separate results folder is provided. The folder name begins with the DUT loop count and includes the DUT ID provided in the TPR Options of the Testplan Editor (default name or user-defined name).
- "LogDutInfo" can be placed in each test procedure phase of all test procedures in order to log the DUT loop information in each result file. Alternatively, it can be placed in the "DUT Loop After" phase which centrally stores the DUT loop information in the `DUTLoopResults.log` file.

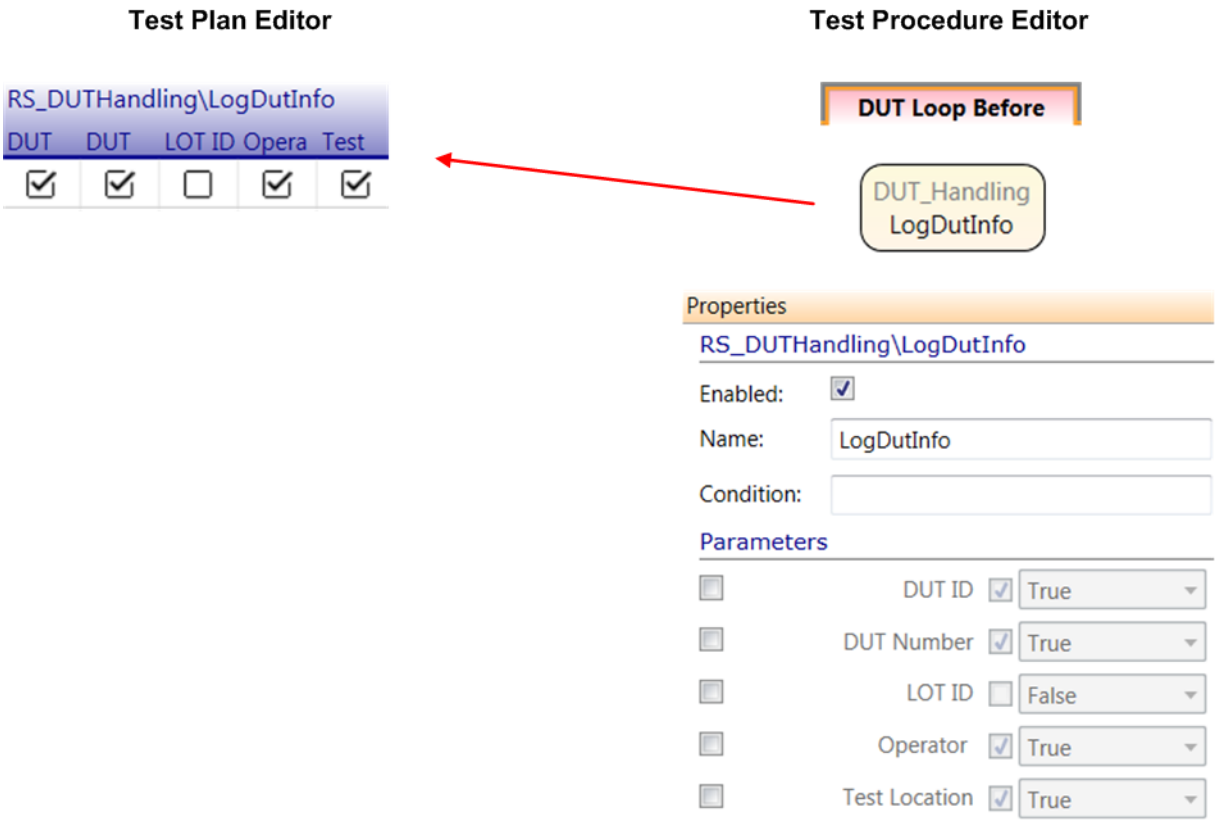


Figure 4-40: Usage of LogDutInfo

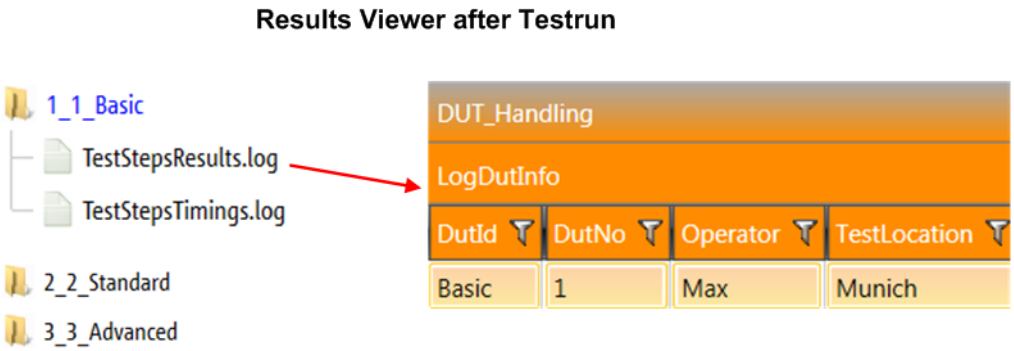


Figure 4-41: LogDutInfo results

4.11 Result Handling

4.11.1 Charts

QuickStep provides means to visualize test results in a chart within an extra window.

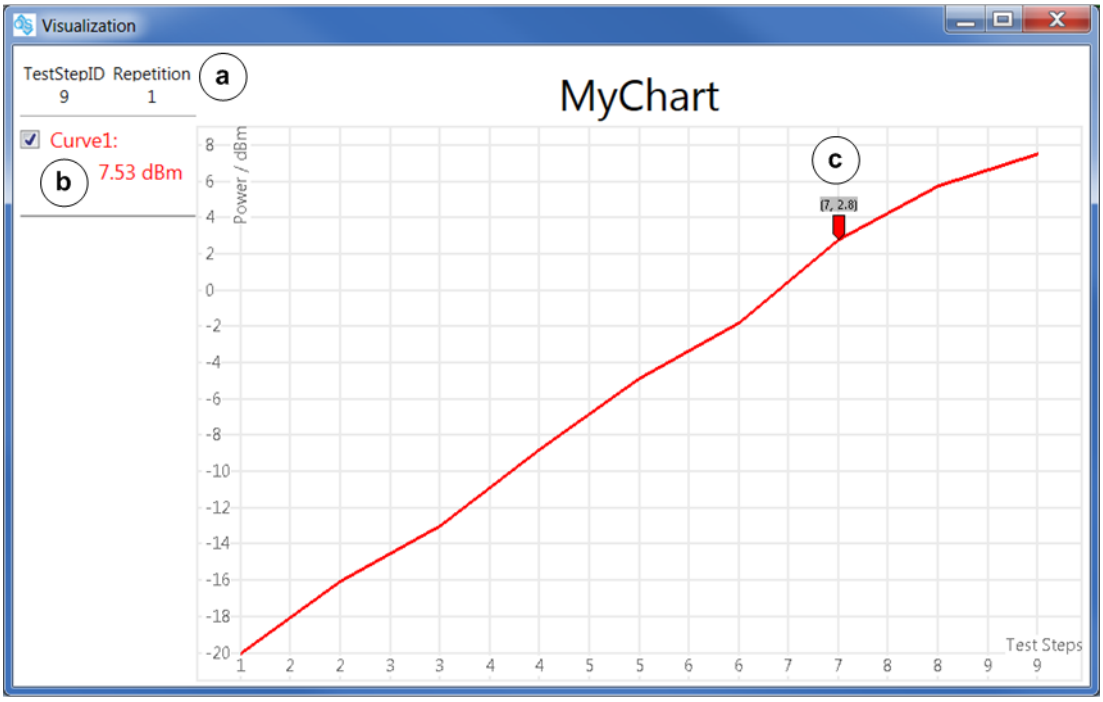


Figure 4-42: Chart window with curve at halt of test run

a = Current Test Step ID and Repetition
b = Result value for the current Test Step ID and Repetition
c = Movable marker showing the data point values

The charts are defined and configured by functions of the "RS_Visualization" block. This is done in the Test Procedure Editor. The visualization block offers different chart types such as cartesian or polar charts, 3D charts or histograms.

For all visualizations of results in a chart, several steps are required realized by block functions as shown below.

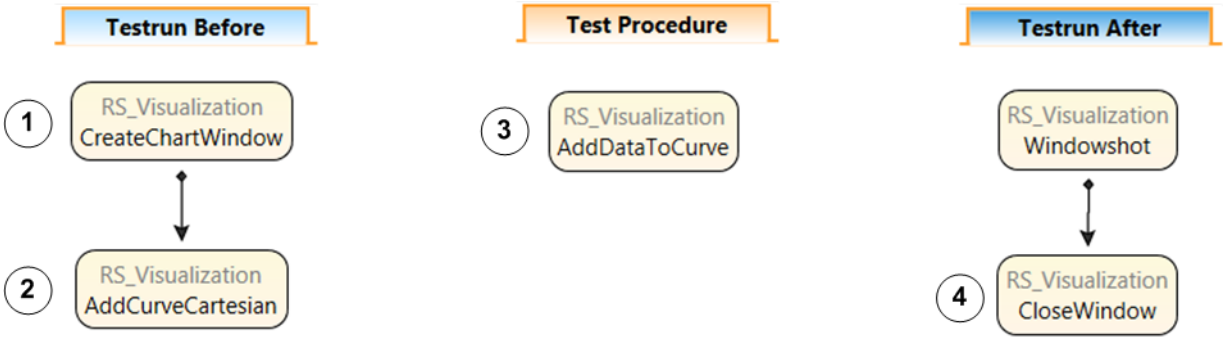


Figure 4-43: Block functions used for creating a chart in an extra window

The numbers in the figure correspond with the visualization steps of the procedure below. The additional "Windowshot" block function is optional and provides a bitmap file of the chart window content before the window is closed.

Preparation step: In the Test Procedure Editor, at "Blocks & Connectivity", add the "RS_Visualization" block (available in the "Utilities" section of the "Library").

1. Create a chart window.
   An additional chart window will be opened during test run. Since it will visualize results over the test steps, the "Create Chart Window" block function is defined outside the "Test Procedure", that is in the "Testrun Before" test execution phase.
2. Add a curve, histogram or another chart element to the chart window.
   The "Add <element> ..." block function is set up as child of the "Create Chart Window" block function in the "Testrun Before" test execution phase.
3. Add data to the curve or other chart element defined in the previous step.
   The related block function is set up in the "Test Procedure" test execution phase since result data is collected in that phase. The block function is connected to the curve of the previous step by the curve ID (analogous for other chart elements). The data values are set by reference to result parameters of the measurement block functions.
4. Close the chart window.
   This action takes place at the end of the test run, that is in the "Testrun After" test execution phase.

The details for the chart window and curve are configured by the properties of the block functions. In the example, the horizontal axis (axis 1) shows the test step numbers, the values in vertical direction are given as results per test step.

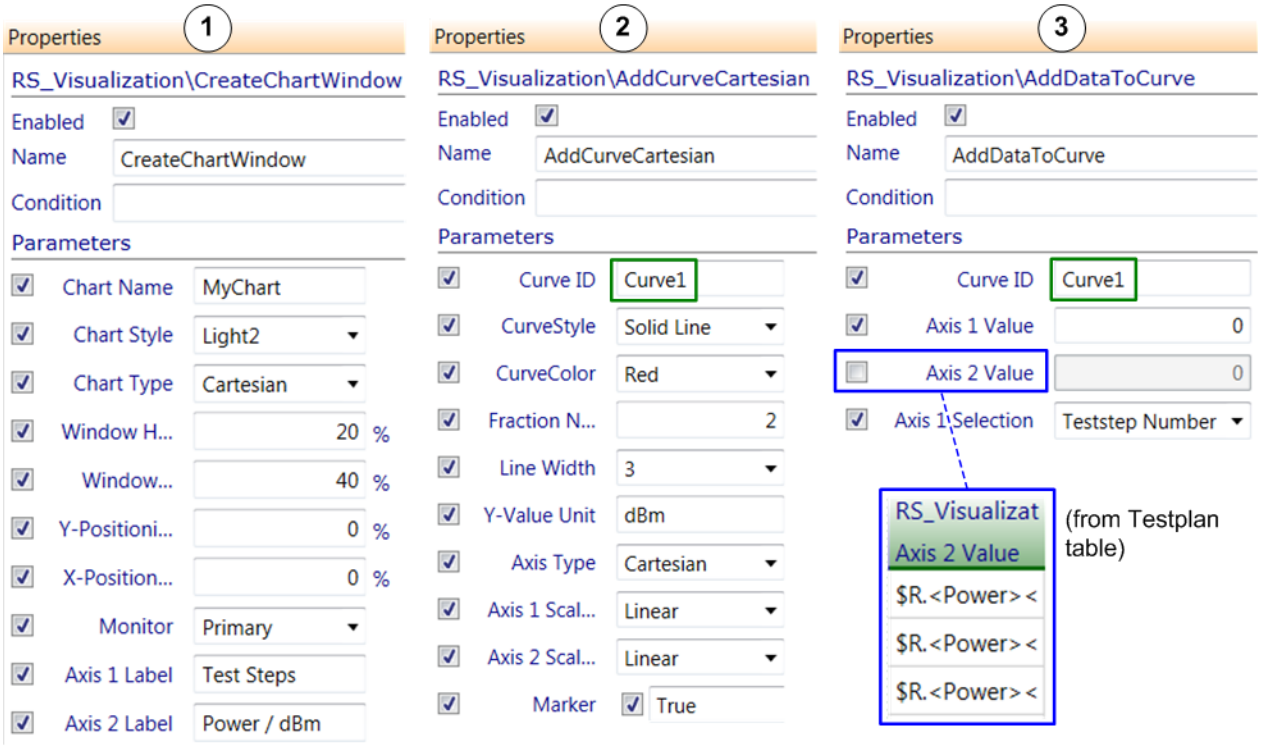


Figure 4-44: Configuration of chart window, curve and data for the curve

Hints:

- Use breakpoints in the Testplan Editor to halt the test execution and inspect a chart in detail.
- Only one "Visualization" window can be opened for one "RS_Visualization" block available in the "Blocks & Connectivity" layer. To enable the creation of a second "Visualization" window, add a second "RS_Visualization" block in the "Blocks & Connectivity" layer. Also add a second "Close Window" block function in the "Testrun After" test execution phase. The QuickStep GUI automatically appends a number suffix to the second "RS_Visualization" block to separate the functions for both blocks.
- More than one curve can be added to a chart window. Take care that the curves have different curve IDs. Otherwise, the test execution is aborted. This behaviour also applies for every other visualization element.
- Only one histogram or surface can be created in one chart window.
- Adding data to a curve relies on the curve ID (analogous for other visualization elements). If the ID does not exist, the test execution is aborted.

4.11.2 Live View Results

The "RS_Visualization" block provides functions for showing current (live) results during test run in an extra window.

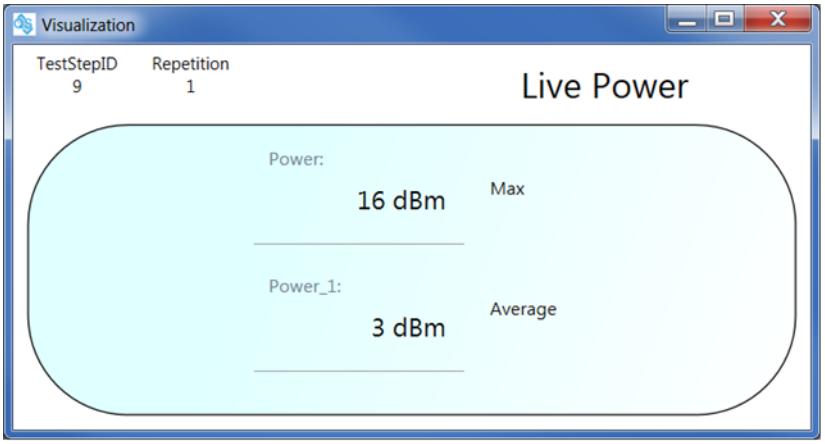


Figure 4-45: Live result visualization window

Creation and configuration of the live view is done with specific block functions which are analogous to those for charts.

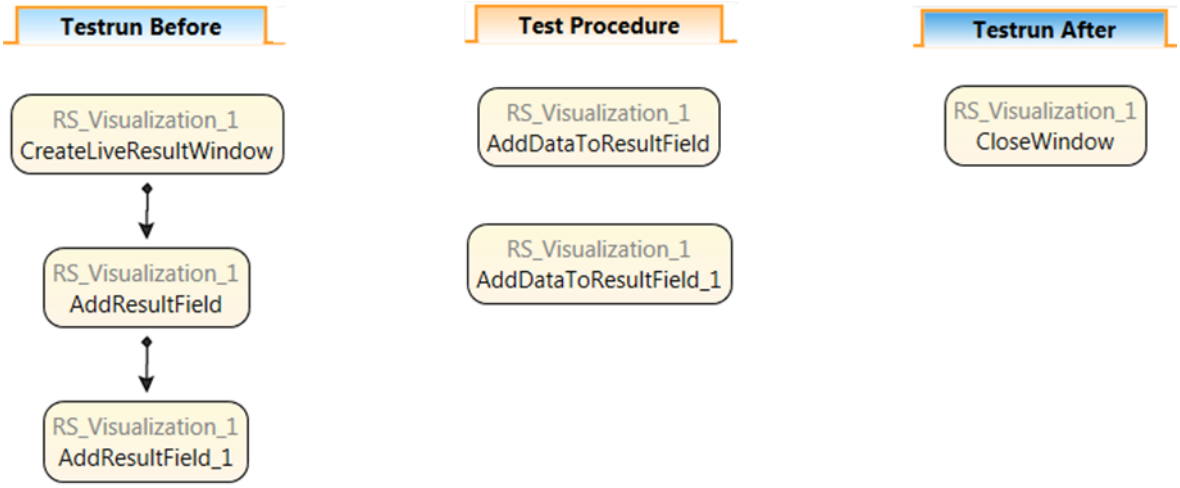


Figure 4-46: Block functions for live result visualization (two results)

The following four steps are required after having added the "RS_Visualization" block in the "Blocks & Connectivity" layer:

1. Create a live result window.
2. Add a result field to the live view window.

3. Add data to the result field defined in the previous step.
4. Close the live result window.

4.11.3 Reports

Reports store test results in doc (Word), pdf or HTML format. The resulting files can be inspected in the Results Viewer but also independently from QuickStep.

The report functionality is realized by the "RS_Report" block. A sequence of block functions has to be set up and configured in the Test Procedure Editor over several test execution phases. The result values to be reported are set by reference to result parameters of the measurement block functions. The following figures show an example report for a test where several DUTs are tested in a row, the related test procedure and block configurations. The numbers indicate corresponding elements in the figures.

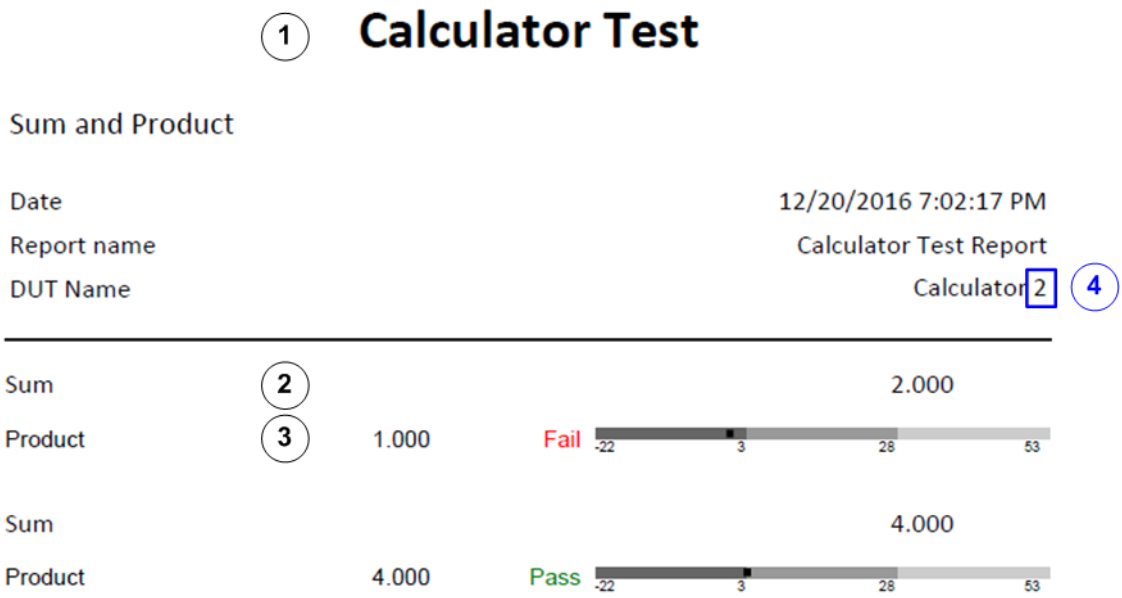


Figure 4-47: pdf report (example)

1 = Header information from the "CreateReport" block function
2 = Result from the "AddResult" block function
3 = Result from the "AddResultMinMax" block function, includes limit handling
4 = DUT number according to DUT loop, automatically inserted

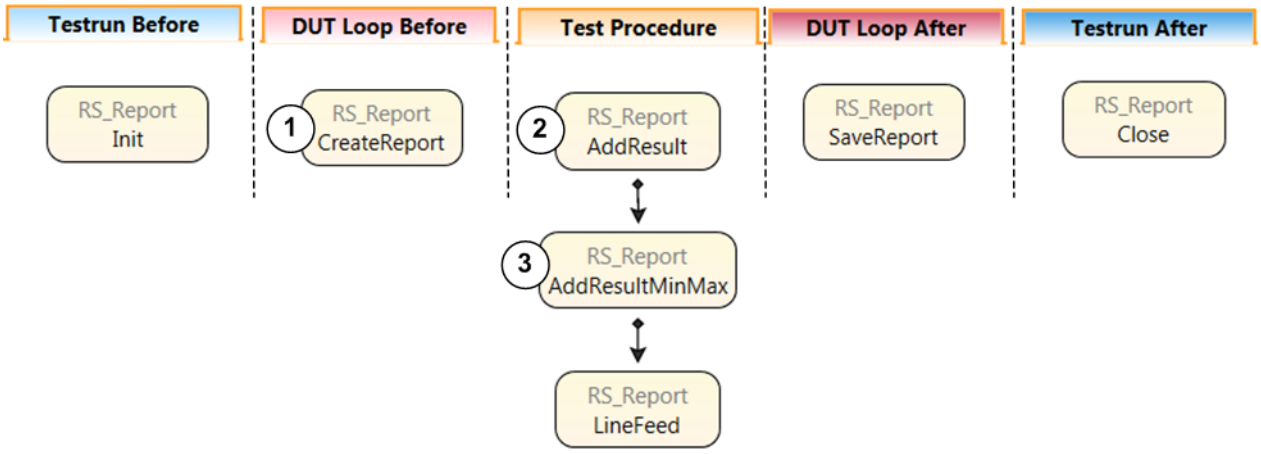


Figure 4-48: Block functions for creating a report (example)

Note that the report is created in the "DUT Loop Before" execution phase and saved in the "DUT Loop After" execution phase. So, a separate report is generated for each DUT (additionally, the "DUT_Handling > AskForDutInfo" block function in the "DUT Loop Before" execution phase is required, test execution with "Continuous Run"). The DUT number as shown on top of the "DUT Information" dialog box during test run is automatically inserted in the current test report.

The "Close" block function is necessary to terminate the block successfully and to complete the queue of report documents.

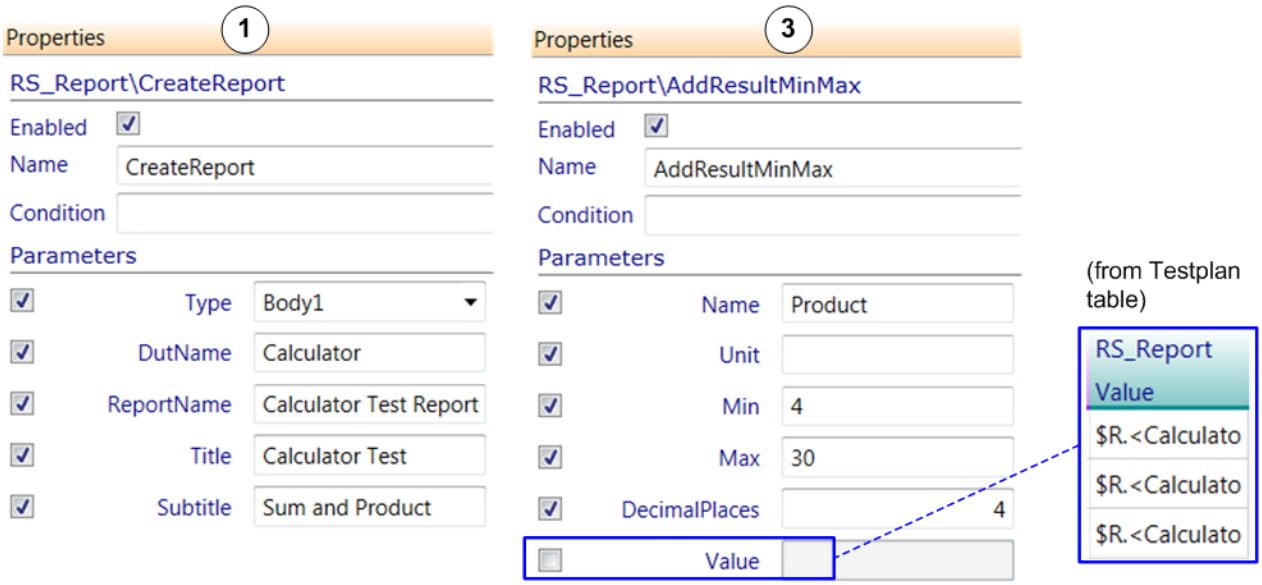
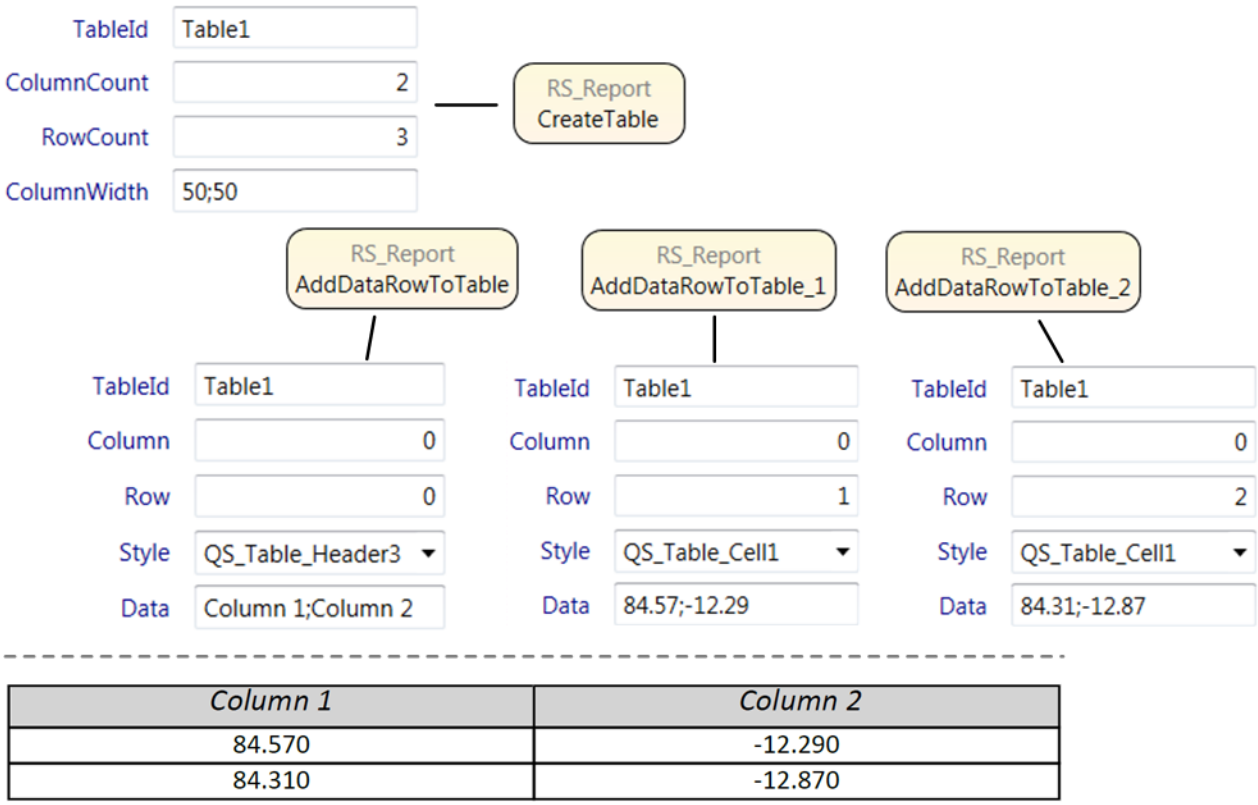


Figure 4-49: Report configuration (example)

The QuickStep reporting functionality is based on ActiveReports® from GrapeCity. The ActiveReports documentation is included in the QuickStep documentation folder accessible via the Windows "Start" menu.

Report with table

The following figure shows by example the block functions and their configurations to include a table in a report and the resulting table.



Column 1	*Column 2*
84.570	-12.290
84.310	-12.870

Figure 4-50: Table in a report

Report with chart from trace file

Data from a trace file can be displayed in a chart within a report. Therefore, the "TraceToChart" block function is provided. The trace file (storing data in character separated columns) is usually created during test run by a test instrument and stored with an appropriate block function (for example "RS_RfSignalAnalyzerBase > SaveTrace").

Figure 4-51: Configuration for displaying trace data as chart in a report

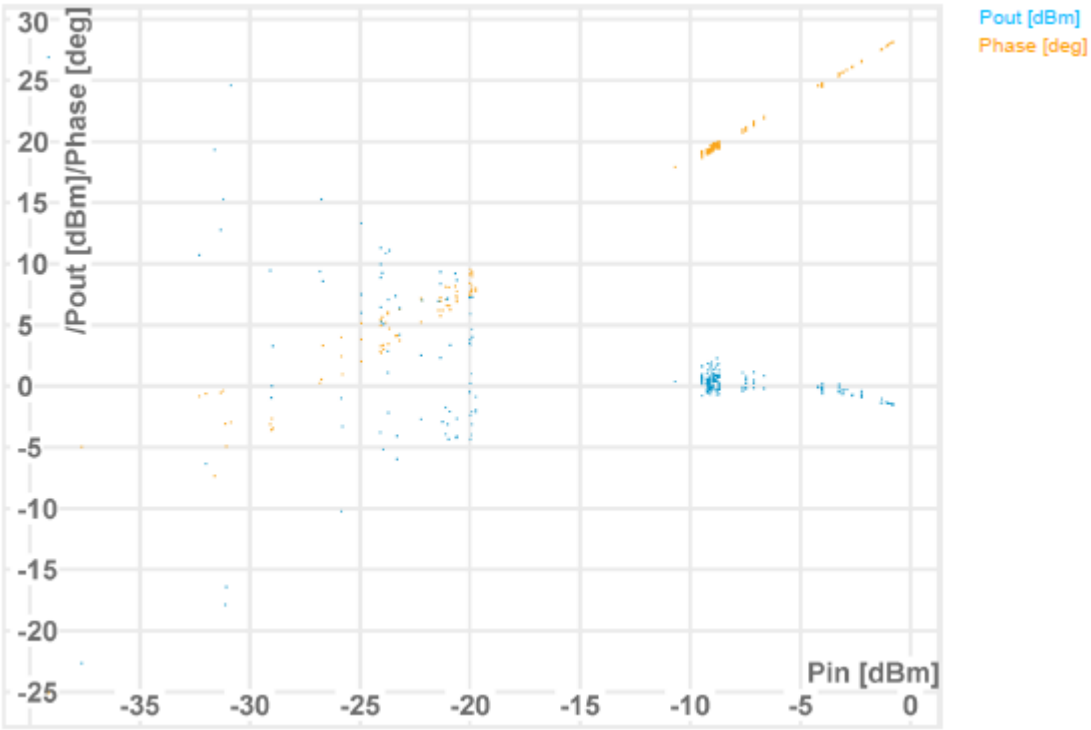


Figure 4-52: Cartesian chart in a report

4.11.3.1 Customizing Reports

QuickStep uses report definitions in RDL format to set up the structure and appearance of its reports. RDL, Report Definition Language, is an XML application for describing reports. RDL report definitions for QuickStep are stored in `.rdlx` and `.rdly-styles` files.

Report customization is done by editing RDL report definitions and connecting QuickStep report block functions to them. For editing report definitions, QuickStep provides the ReportDesigner tool accessible via the Test Procedure Editor's tool-

bar. The following figure shows an example scheme of the relations. Advanced users can use an XML editor for modifying existing RDL files.

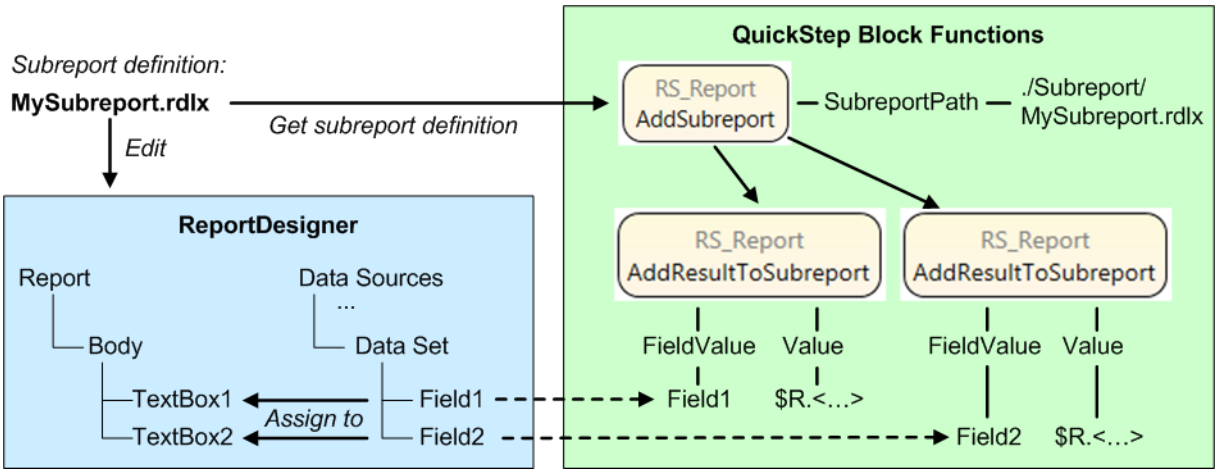


Figure 4-53: Relations between .rdlx file, ReportDesigner and block functions (example)

The report definitions used by QuickStep are distributed in three components each of them with their own RDL files:

- Subreport (`.rdlx` files)
- Header (`.rdlx` files)
- Style (`.rdly-styles` files)

See the following sections for details.

It is recommended to collect commonly used `.rdlx` files under `C:\Users\Public\Documents\Rohde-Schwarz\QuickStep\Common\ActiveReports\` and the `Subreport` or `Header` subfolder.

User defined `.rdly-styles` files have always to be stored under `C:\Users\Public\Documents\Rohde-Schwarz\QuickStep\Common\ActiveReports\Styles`.

Conventions for paths

Some block functions need the path and filename for the RDL file to be used. Absolute and relative paths can be entered.

Absolute paths are entered according to the Windows standard notation, for example `C:/folder/file.rdlx`.

Relative paths must start with either "`./`" or "`../`". "`./`" indicates that the file is in the same directory as the test plan. "`../`" indicates that the file is located in the parent directory.

Subreports

A report can contain different configurable report parts in its content area which are called subreports.

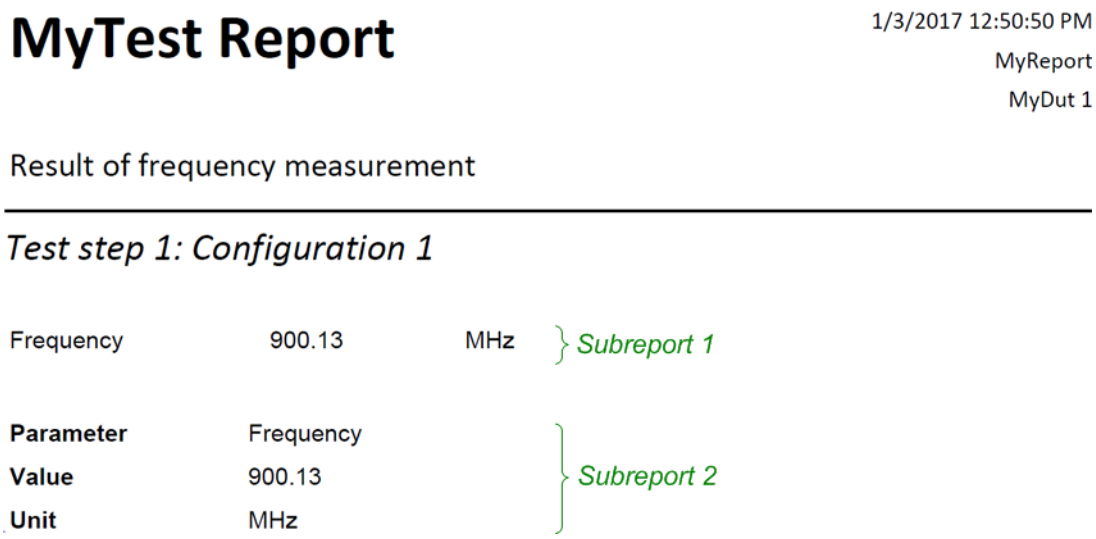


Figure 4-54: Example subreports within a report

First, the structures of the subreports have to be defined in `.rdlx` files. Afterwards, these files are used by QuickStep block functions. The following figures show how the RDL structures of the example subreports are displayed in the ReportDesigner. The subreport definitions have fields serving as containers for data from QuickStep. The fields are assigned to text boxes which define positions, sizes and typography. Additional elements such as text or tables can be included.

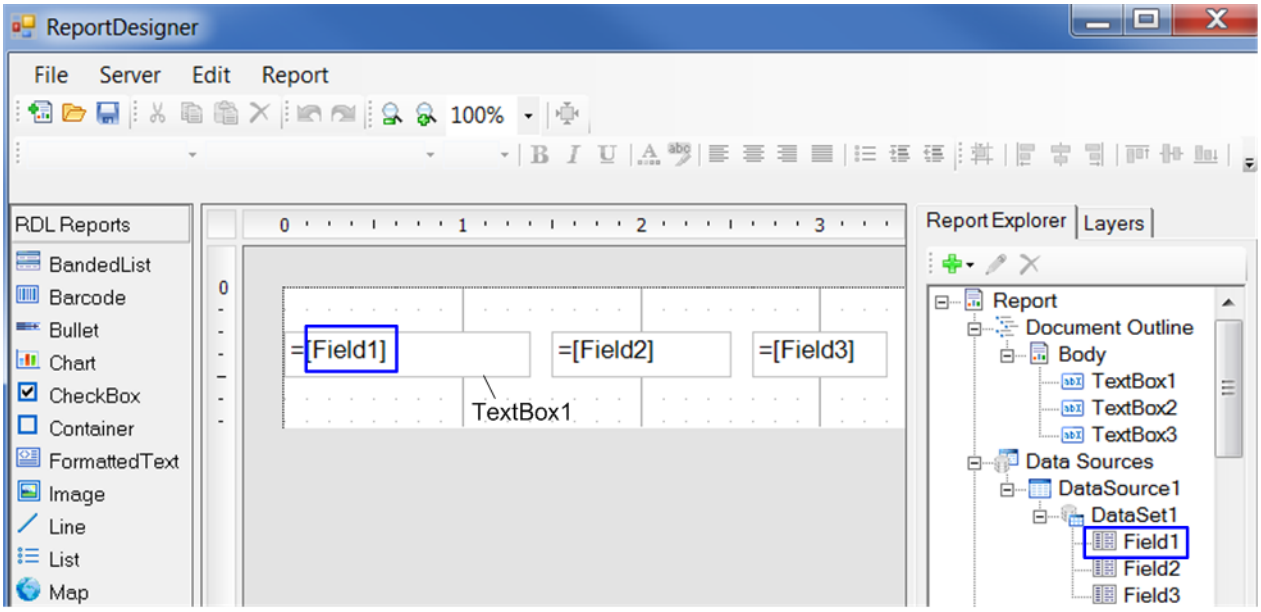


Figure 4-55: Subreport 1 definition in the ReportDesigner

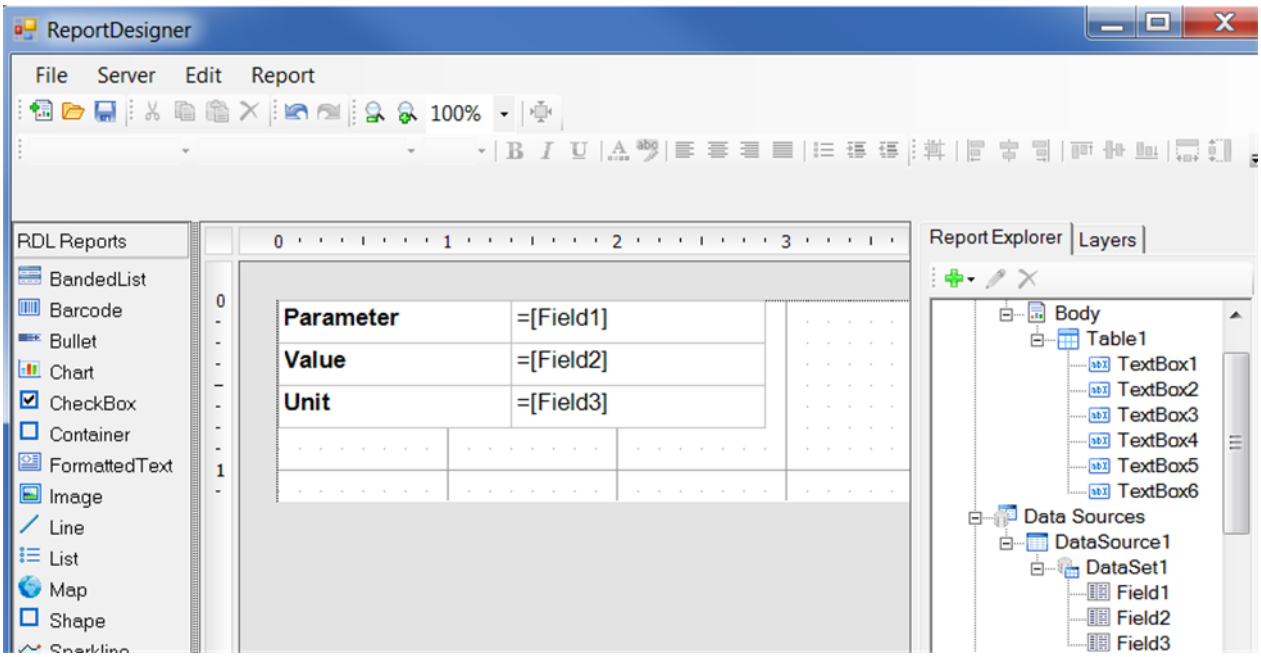


Figure 4-56: Subreport 2 definition in the ReportDesigner

In QuickStep, the "RS_Report" block provides the required subreport functions, see the example test procedure in the following figures. The numbers in the figures indicate corresponding parts.

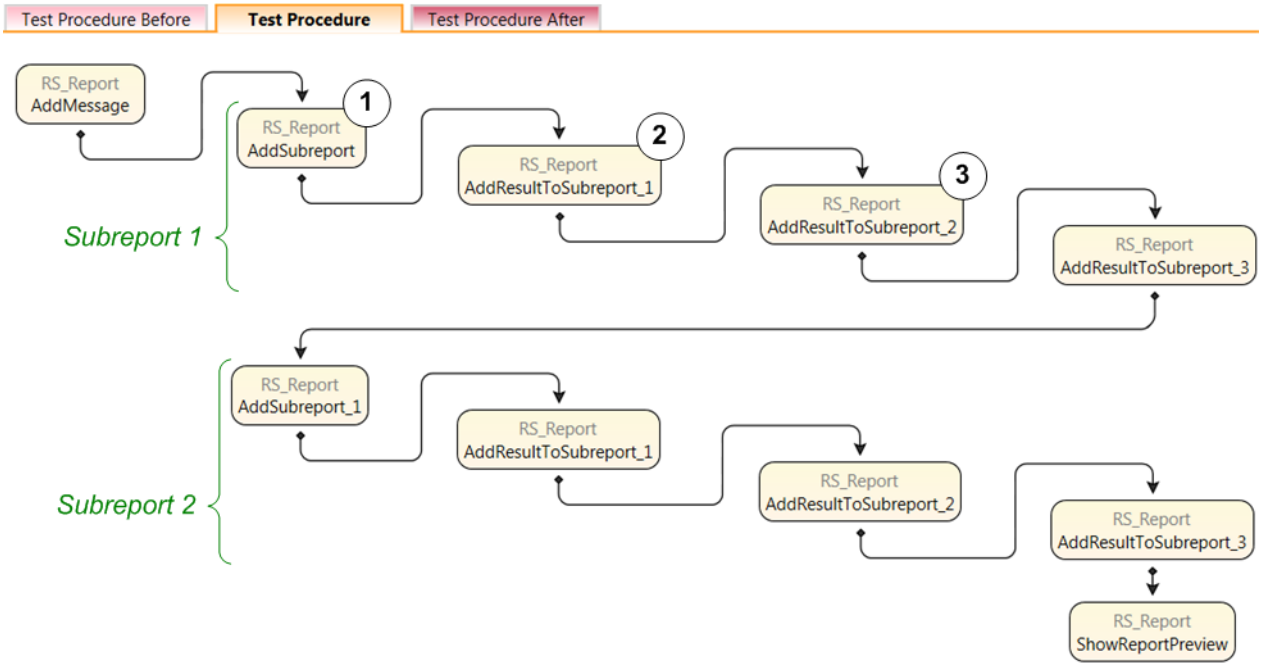


Figure 4-57: Test procedure for two subreports

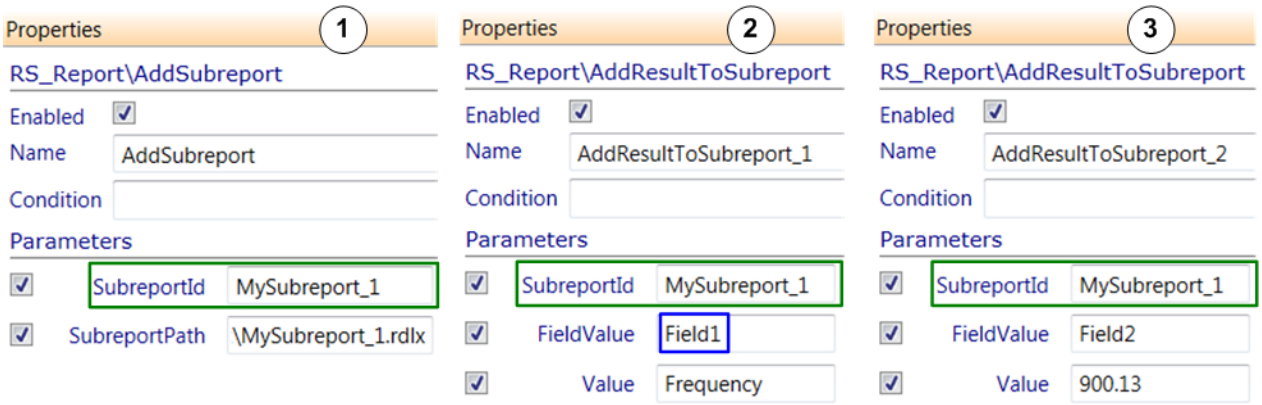


Figure 4-58: Block function properties for subreport 1

The "AddSubreport" block function adds a subreport and connects to the `.rdlx` subreport definition given in the "SubreportPath" variable. Each "AddResultToSubreport" block function fills a field of the subreport definition with data by assigning a "Value" to a "FieldValue" (field name of the subreport definition). The "AddResultToSubreport" block functions are connected to the "AddSubreport" block function by the "SubreportId".

Configuration of the Report Header

The report header can be customized with an `.rdlx` header definition. See the following figures for an example.

Figure 4-59: Customized header in pdf report

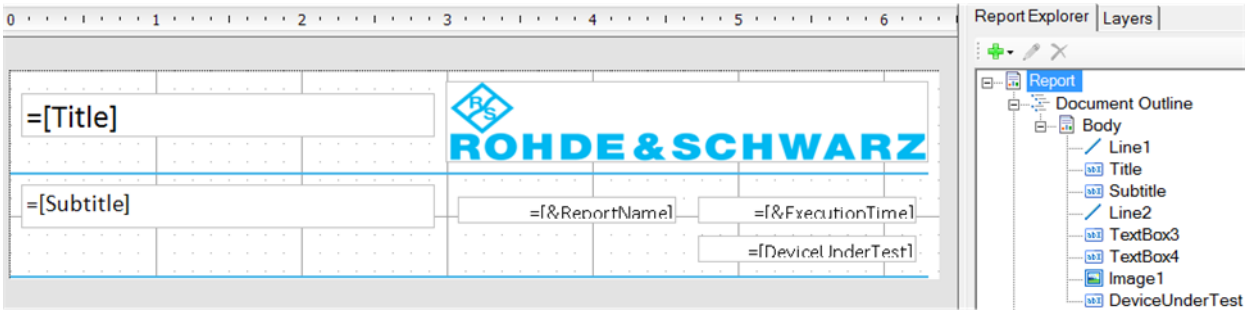


Figure 4-60: Customized report header definition

In the RDL header definition, the fields which can be filled by QuickStep must have the following names:

- Title
- Subtitle
- DeviceUnderTest

The "CreateReport" function of the "RS_Report" block connects to the RDL header definition. If the fields in the RDL header definition have the names listed above, they are filled with the values of the corresponding parameter values in the "CreateReport" block function.

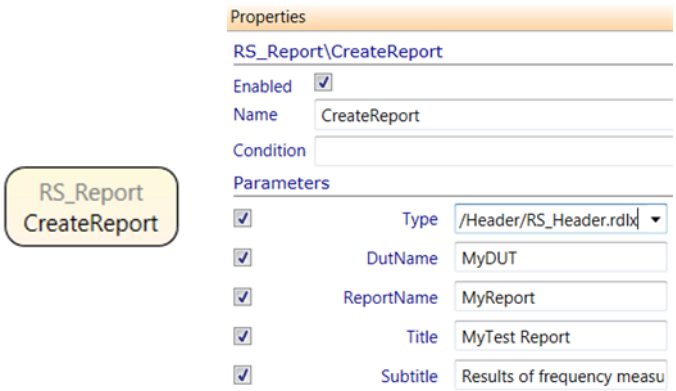


Figure 4-61: Connecting to an RDL header

Report Styles

Some report block functions which refer to certain report elements assign a style to the report elements. For example, the "AddResult" block function has a "Style" variable whose value determines the font and other properties of the added result. QuickStep provides several styles which can be selected at the block function's "Style" variable. A user-defined style can also be assigned by entering its name directly. The user-defined styles have to be defined in `.rdly-styles` files and are stored under the following path:

`C:\Users\Public\Documents\Rohde-Schwarz\QuickStep\Common\ActiveReports\Styles`.

QuickStep automatically scans the `.rdly-styles` files under this path for user-defined styles if necessary.

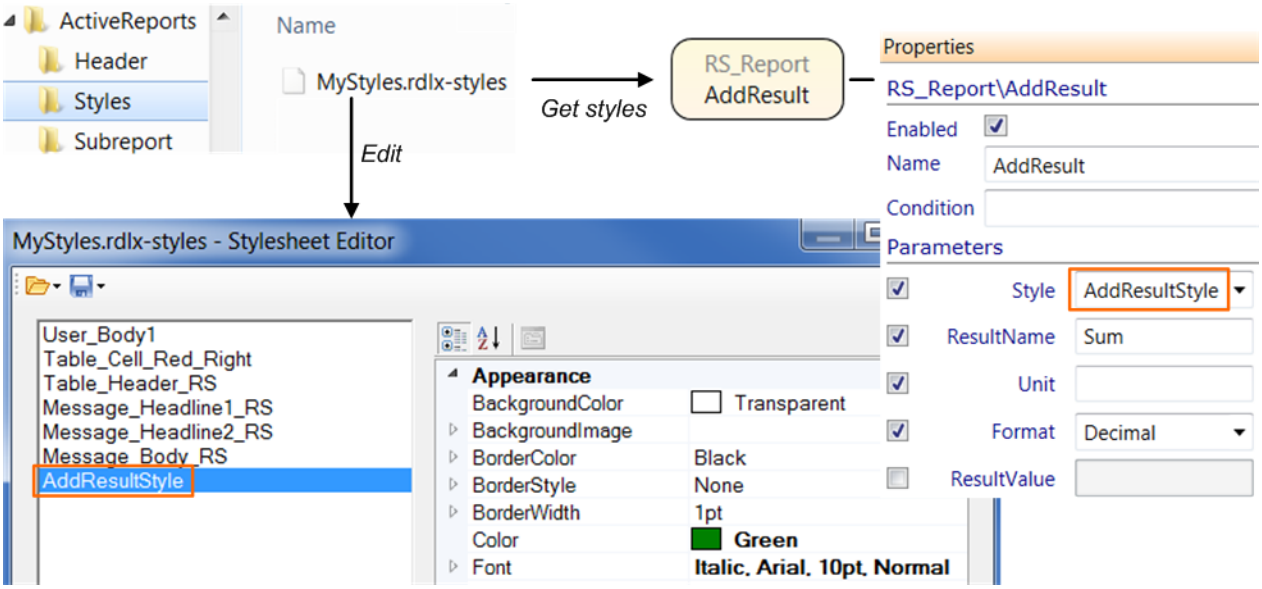


Figure 4-62: Editing and assigning user-defined styles

Figure 4-63: Report result with user-defined style

User-defined styles in an `.rdly-styles` file are edited with the Stylesheet Editor within the ReportDesigner. This editor is accessible from the ReportDesigner's "Report > Stylesheet Editor" menu.

5 Preparing for Use

5.1 Required Hardware, Software and Firmware

Required hardware and operating system

- Standard PC
- Windows 7 as 64 bit version, including service pack 1 and universal C runtime update (KB2999226) or Windows 8.1, Windows 10

Required software

The QuickStep installer includes all required software. That is the QuickStep software itself and environment software such as Microsoft C++ Redistributable Packages, .NET Framework including Developer Pack, and the R&S License Server Manager including the R&S License Server.

Additional and Optional software

If you want to use VISA instruments for your tests or even develop your own QuickStep blocks, then install the related additional software.

Table 5-1: Additional and optional software

Software	Usage
R&S VISA 5.5.5 or higher	VISA communication with test instruments (except for GPIB)
VISA for GPIB instruments provided by the manufacturer	Alternative VISA communication with test instruments for GPIB connections
NI Driver for GPIB-USB-HS adapter	NI USB-GPIB adapter
R&S NRP Toolkit	NRP power meter measurements; includes NRPZ VXI plug&play drivers
SignalCraft Scout Driver	SignalCraft Scout USB to serial / GPIO adapter
Microsoft Visual Studio (Community) for C++/C# (supported versions: 2012, 2013, 2015, 2017; for C++ also 2010)	Integrated development environment for block development

Software	Usage
R&S Forum (supported versions: 3.2.0, 3.3.0)	Editor and runtime environment to create and execute Python scripts, for example for remote control of instruments
Mathworks MATLAB (supported versions: 2016b, 2017a)	Editor and runtime environment to create and execute MATLAB scripts

R&S-VISA and NI-VISA can be installed in parallel. The VISA variant can be selected per connection type (GPIB, Socket, HiSLIP, USB), for example in the RsVisaConfigure tool. R&S VISA does not support GPIB connections.

The optional software is not provided with QuickStep but can be downloaded from the manufacturer's website or is provided together with the test instruments.

For obtaining Visual Studio Community, search for "Visual Studio Community" on https://www.microsoft.com/en-us/download and follow the first link.

Special test instruments may require additional software on the host PC, for example device drivers.

Required firmware on the test instruments

The provided example test applications require certain firmware versions and options of the involved measurement equipment. So, verify that the firmware versions and options on the measurement equipment comply with the requirements given in the release notes.

Refer to the manuals of the measurement equipment on how to check and respectively update the firmware if necessary.

5.2 Installation

5.2.1 Putting the Smart Card into Operation

The QuickStep software is licensed with a smart card-based licensing system. The smart card in SIM format can be used together with the card reader provided with the QuickStep delivery. Alternatively, the full format smart card can be inserted into a reader connected to, or built into, a PC.

A smart card and a USB smart card reader (USB stick) is provided with the delivery.

Proceed as follows:

1. Break out your smart card.

Figure 5-1: Smart card and USB smart card reader

2. Insert the smart card with the chip facing upwards and the angled corner facing to the USB stick into the USB stick. The USB stick's LED or "Rohde & Schwarz" label is also facing upwards.

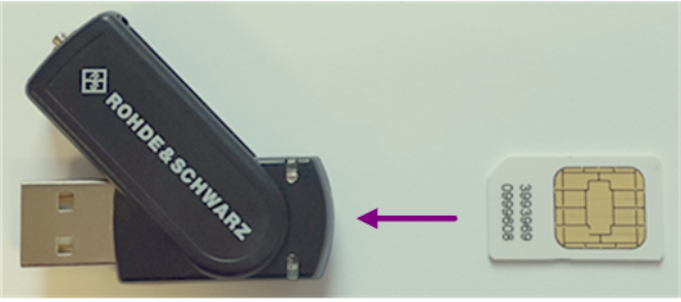


Figure 5-2: Inserting the smart card

3. Push the smart card completely inside the USB smart card reader.

The QuickStep software can now be used together with the USB smart card reader.

5.2.2 Installing QuickStep

The QuickStep software is provided as `RSQuickStep_[BuildNo].exe` file.

Proceed as follows to install QuickStep on the PC:

1. Copy the `RSQuickStep_[BuildNo].exe` file into a local directory on your PC.
2. Double-click `RSQuickStep_[BuildNo].exe`.
   The installation wizard opens. The dialog includes the license terms and conditions.

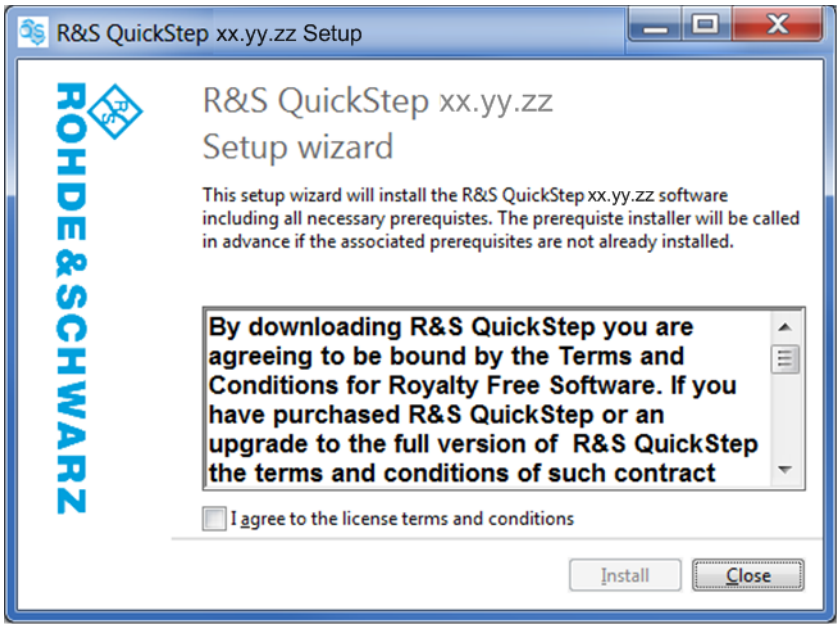


Figure 5-3: Installation wizard

3. Read the license terms and conditions carefully and tick the checkbox at "I agree to the license terms and conditions" if you agree. Then click "Install" to start the installation.
   The installation progress is displayed. The installation comprises QuickStep itself, environmental software and QuickStep applications.
   The "Installation Successfully Completed" report indicates the end of a successfull installation process. Additionally, optional software packages not installed yet are listed (to be installed later if required for your intended usage of QuickStep).
   The installation of the environmental software might require a restart. Click "Restart" in this case to finish the installation.
4. Click "Launch" or "Close".

QuickStep is stored under
`C:\Program Files\Rohde-Schwarz\QuickStep\`. The example and user

files can be found under `C:\Users\Public\Documents\Rohde-Schwarz\QuickStep`. Depending on the Windows installation this path might vary. Use the links in the "R&S QuickStep" folder in the Windows start menu to determine the exact location.

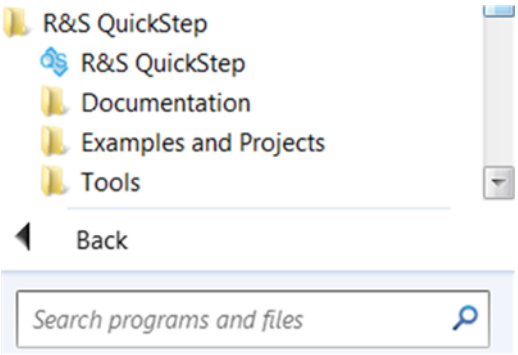


Figure 5-4: QuickStep in the Windows start menu

Separate icons are created on the desktop to directly start QuickStep, the QuickStep Block Development Tool and the QuickStep OTA application.

If Forum 3.2.0 is installed on the PC, QuickStep copies a Python file into the Forum installation directory to enable the QuickStep-Forum integration. This might cause a Windows “User Account-Control” warning depending on the setup of the PC. Administrator rights are required to copy this file.

QuickStep is ready to be started at the PC.

Installation via command line with log file creation

In case of an error during usual installation, try the installation via command line with log file creation:

1. Open a command shell.
2. Navigate to the directory containing the QuickStep `*.exe` installation file.
3. Enter *RSQuickStep_<Version>.exe -log quickstep_installation.log*.
4. If the installation still ends with an error, send the created `quickstep_installation.log` via e-mail to R&S Customer Support to get further help. Use the contact information listed at http://www.customersupport.rohde-schwarz.com.

5.2.3 Installing Optional Software and Firmware

- Get the optional software packages as needed for your purposes, for example VISA.
- Install the optional software packages.
  - For remote control of VISA instruments, a VISA library is required as prerequisite.
  - If you use the SignalCraft Scout USB to serial / GPIO adapter, see Installing the SignalCraft Scout Driver.
- For block development with C++ or C#, install Microsoft Visual Studio – at least Visual Studio Community. For obtaining Visual Studio Community 2017, search for "Visual Studio Community 2017" on https://www.microsoft.com/en-us/download and follow the first link.
- Install required firmware on your test instruments if needed. For requirements and instructions see the user manuals for the test instruments.

5.2.4 Updating License Keys

The QuickStep software options are enabled by license keys (also called option keys). In case of an update of an existing license dongle, the license keys are provided as key codes. The license keys are assigned to the unique serial number of the QuickStep smart card. An update of license keys is done with the R&S License Server Manager. The R&S License Server Manager shows currently available licenses and allows to administer the licenses on the smart card. The R&S License Server Manager is part of the QuickStep delivery.

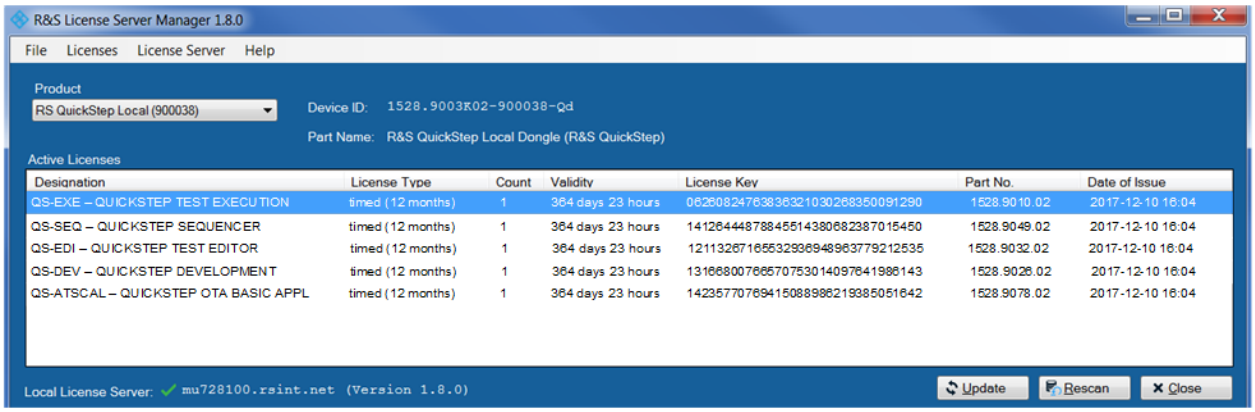


Figure 5-5: R&S License Server Manager

The R&S License Server Manager provides an integrated help. See this help for detailed information.

To add a license key

Preparations:

- Make sure that QuickStep is installed on the same PC which is used for the license key update. Otherwise the R&S License Server Manager does not display the “Designations” of the license keys.
- Make sure that the USB reader with the QuickStep smart card (carrying the serial number) is connected to the PC.

1. Start the R&S License Server Manager via Windows "Start" menu.
   The R&S License Server Manager detects the smart card and lists all active license keys.
2. Select "Licenses > Activate License" from the menu.
   The "License Activation" dialog is opened.

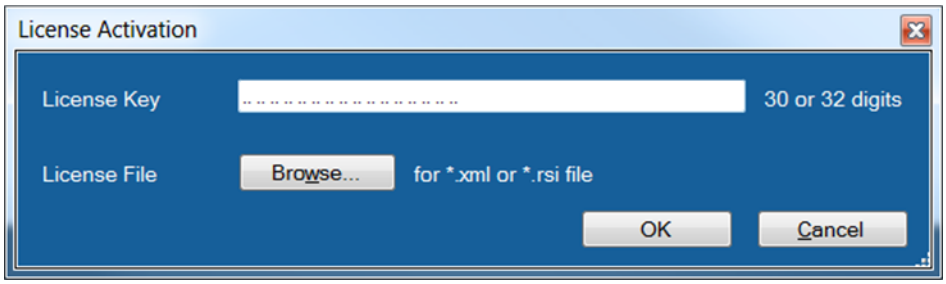


Figure 5-6: License activation

3. Enter the provided license key code or select a license file via "Browse...".
4. Click "OK".

5.3 Network Configuration

The PC hosting QuickStep and the test instruments usually operate within a closed network. Therefore, an IP configuration is required on the host PC and the test instruments. The PC's and the test instrument's addresses must be unique and must belong to the same IP subnetwork.

IP configuration on the host PC if a DHCP service is not available

Proceed as follows on the host PC:

1. Start from the Windows "Start" button, select the "Control Panel", then "Network and Internet" and "Network and Sharing Center".
2. Click the desired connection in the "View your active networks" section.

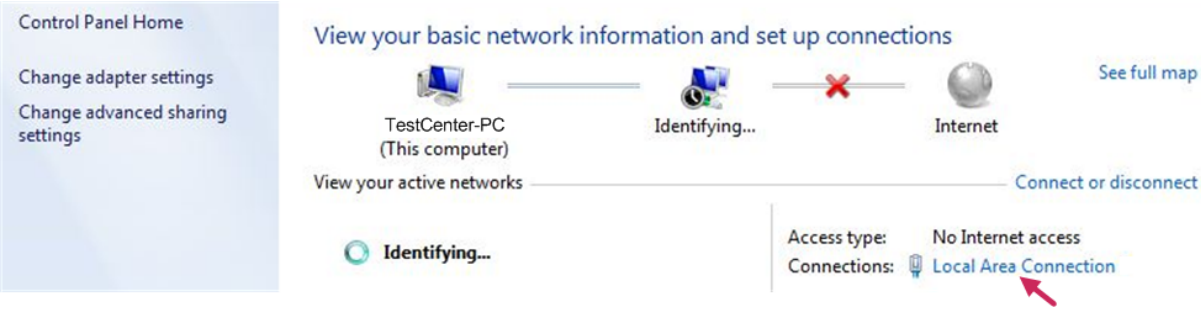


Figure 5-7: Network selection

3. Click at "Internet Protocol Version 4 (TCP/IPv4)" to highlight it and then click the "Properties" button.

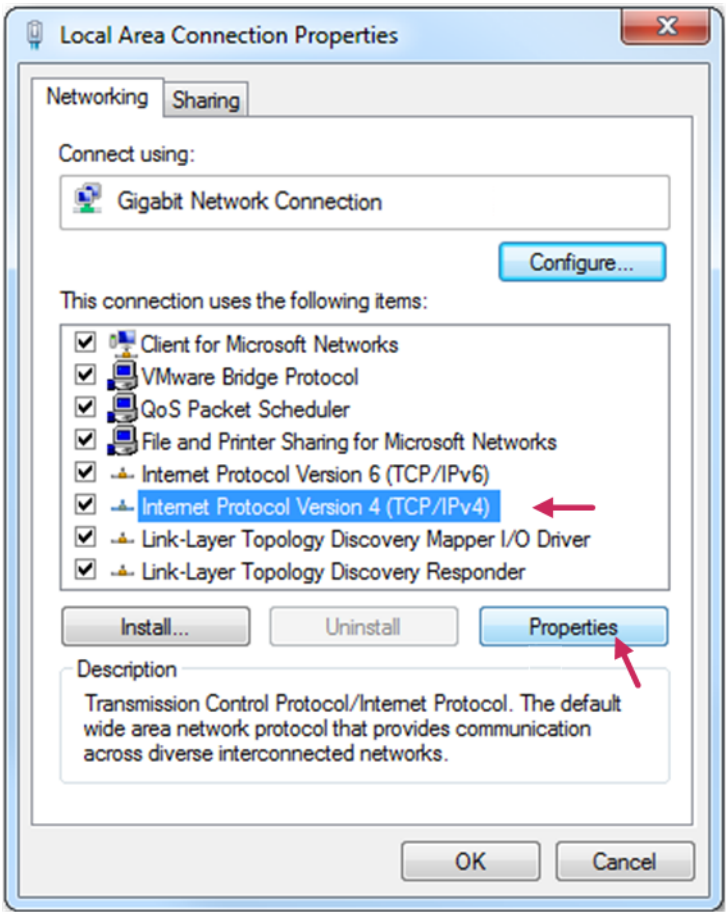


Figure 5-8: IPv4 properties

4. Enter an unused IP address within the subnet of the network used for testing and use an appropriate subnet mask.

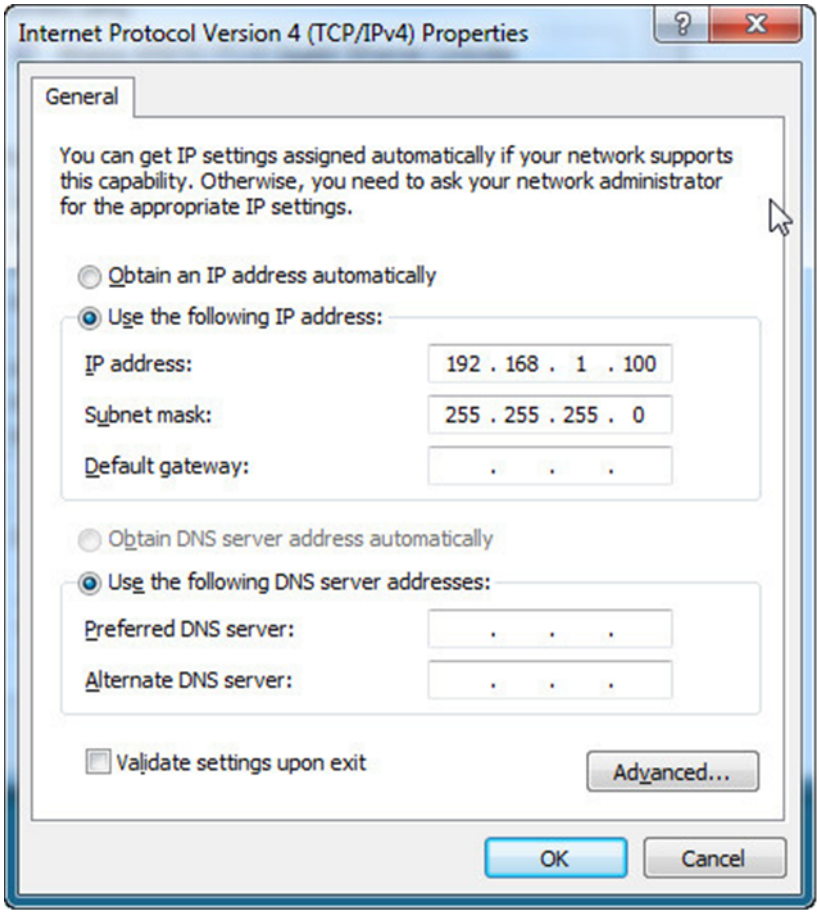


Figure 5-9: IP address and subnet mask

5. Click "OK".

IP configuration on the test instruments

Please refer to the manuals of the measurement equipment on how to change the IP addresses. Take care that each test instrument and the host PC are assigned to different IP addresses within the same IP subnet.

5.4 Determining Input and Output Attenuations

A very basic way to determine the attenuations of a connection or a RF device is to apply a continuous wave signal with fixed RF frequency and RF level on the relevant connections and measuring the power level at the end of the connections. One measurement is needed for the in attenuation, a second one for the out attenuation (no measurements with a frequency sweep). The RF signals are

generated with a manually configured signal generator. The RF power level measurements are done with a signal analyzer (or with the NRP power meter and the additionally available Power Viewer Software from the Rohde & Schwarz web page).

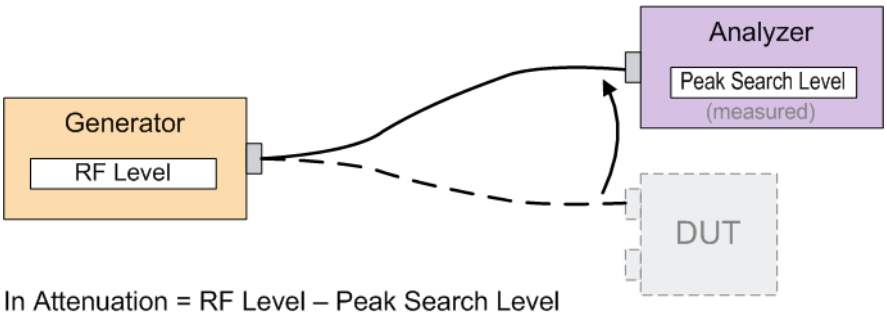


Figure 5-10: Determining the in attenuation

To determine the attenuation for a connection:

1. Connect the generator to the analyzer using just the cable connection you want to evaluate – including the attenuators if part of the connection.
2. At the generator, set the RF Level to 0 dBm.
3. At the generator, set the target RF frequency.
4. At the analyzer, set the mid frequency to the target RF frequency.
5. Run “Peak Search” at the analyzer and read the measurement value.
6. Calculate the attenuation value:
   Attenuation in dB = RF Level (Generator) in dBm – Peak Search Level (Analyzer) in dBm

A better accuracy is achieved if only relative measurements are executed with the analyzer. Alternatively a power meter or a network analyzer can be used.

6 Operation

This chapter provides procedures for configuring, running and evaluating tests. The descriptions begin with simple tasks and step forward to increasing complexity. You can combine procedures or parts of procedures to cover all operational tasks.

You can also refer to the QuickStep Training manuals for step-by-step descriptions of many operational tasks. The manuals are accessible via the Windows "Start" button and the folder "R&S QuickStep > Help and Manuals".

6.1 Starting QuickStep

Prerequisites:

- QuickStep has been installed on the PC.
- A smart card reader with inserted smart card for QuickStep (providing the required licenses) has been connected with the PC. Connecting the smart card can also be done after start of QuickStep.
- The test instruments and the PC have been connected according to the desired test setup.

Proceed as follows:

1. Start all involved test instruments and devices.
2. Ensure that all test instruments are reachable from the PC.
3. At the PC, start QuickStep with a double-click from the Windows start menu or from the desktop icon.

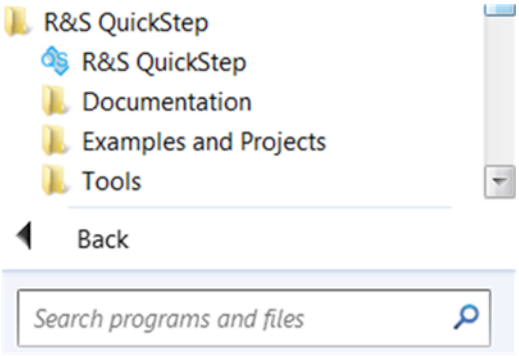


Figure 6-1: QuickStep in the Windows start menu

QuickStep starts up and provides a start dialog.

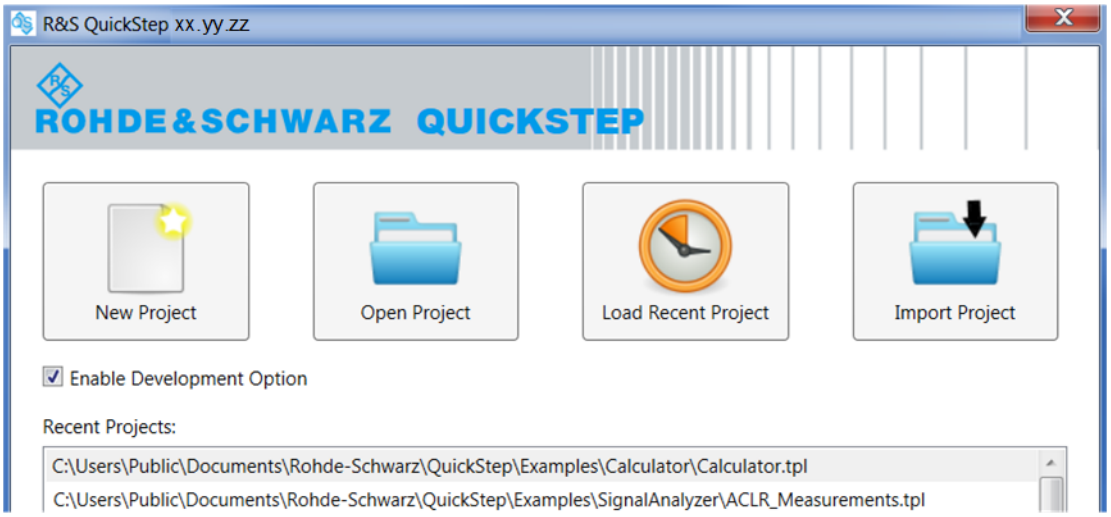


Figure 6-2: Start dialog

If you start the QuickStep OTA ATSCAL application, this start dialog is not shown. Login as user *Operator*. No password is required and the following steps are not relevant.

4. If you do not want QuickStep to occupy an existing QS-DEV license, deactivate the "Enable Development Option". Then, the Test Procedure Editor and the Test Procedure Debugger will not be available.

5. Select one of the project options (for example, "Open Project") or double-click a project from the list of "Recent Projects".
   This selection does not cut the access to other projects. Afterwards, you can change the project or define a new one via the QuickStep "File" menu.

   A license check is performed. Its result is reported in the "Log Viewer".

Recommended next step for beginners

If you are not yet familiar with QuickStep and look for a tutorial, proceed with the training manuals. They are accessible via the Windows "Start" button and the folder "R&S QuickStep > Documentation" or via the Help menu in QuickStep. The training manuals provide step-by-step procedures based on instructive examples.

6.2 Executing a Test Project

Use case: Your test setup and test purpose match a provided test project. Hence, only the connection parameters for communication of QuickStep with the test instruments have to be set and the test plan has to be optimized. A system configuration is not used here. The connection parameters are set in the "Test Procedure Editor" (not in the "System Configurator" which is also possible, see Chapter 6.9, "Using System Configuration Parameters", on page 107).

Starting situation:

- The QuickStep application has been started.
- The test instruments have been connected with the PC where QuickStep is running.
- You know the IP addresses / GPIB addresses of the test instruments and the type of connection.

1. Select "File > Open Test Project" from the top bar menu of the QuickStep GUI. A Windows Explorer opens.
2. Browse to the desired `*.tpl` test plan file and open it.
   The test plan, the related test procedure and system configuration (if available) are loaded.
3. Select the "Test Procedure Editor" and then the "Testrun Before" item in the "Test Procedure Browser".
4. Step through the Init blocks for the test instruments. For each test instrument, enter the "VISA Resource" string (containing the instrument's address): Select it from the preconfigured VISA-alias drop-down list or create the resource string using the "VISA" button next to the parameter.
5. In the "Testplan Editor", click "Update Test Project" in the toolbar to communicate the test procedure changes to the "Testplan Editor".

6. Adjust the sequence of test steps according to your needs:
   - Append test steps with the "Add Test Step" button at the toolbar.
   - Add a parameter sweep, see Chapter 6.3, "Adding a Parameter Sweep to a Test Plan", on page 95.
   - Adjust the parameter values in the table (for example RF level, frequency).
   - Optionally, add a repetition by entering the number of repetitions in the "TPR Options" tab on the right side.
     Repetitions are useful for statistical evaluations.
7. Save the test project via "File > Save Test Project" on the top menu.
   The test plan is saved in a `*.tpl` file.
8. Still in the "Testplan Editor", start the test plan execution via the "Single Run" button on the toolbar.
   Alternatively, start a continuous execution via the "Continuous Run" button.
   This method is especially usefull to test several DUTs in a row.

Result: The sequence of test steps is executed and for all steps (and their repetitions) the functions given by the connected test procedure are applied. The progress of the test run is indicated in a progress bar. By default, the results are stored in the subfolder `..\Results\` of the corresponding project folder or in the folder defined in the "LoggingPath" parameter in the "TPR Options" tab.

Variation: You can exclude a parameter from the test plan (to reduce the number of columns):

1. Select the "Test Procedure Editor".
2. Select the block function in the main view which the parameter is related to.
   You can see the related parameters in the "Properties" view.
3. Activate the parameter via the check box on the left.
   The parameter becomes editable and is not shown in the "Test Plan Editor" view anymore.

6.3 Adding a Parameter Sweep to a Test Plan

An already available test plan is extended by adding a parameter sweep. For example, the test plan has only one test step (one row) at first. The parameter sweep extends the test plan to 20 test steps.

1. In the "Testplan Editor" view, right-click the row in the test plan table where the sweep shall begin.
   The context menu opens.
2. Select "Sweep Value" from the context menu.
   The "Edit Test Step" dialog opens.

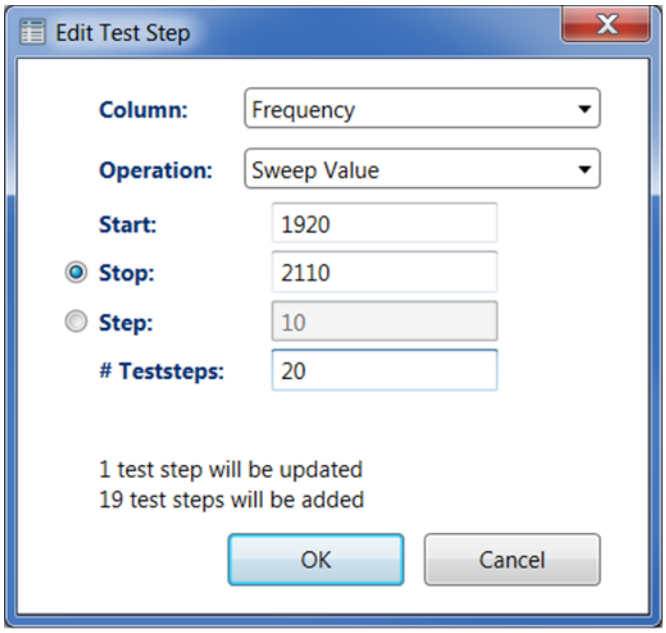


Figure 6-3: Edit test step dialog

3. At "Column", select the sweeping parameter.
4. At "Count", set the number of steps for the complete sweep including the start step.
5. At "Start", enter the desired start value for the sweeping parameter.
6. - Either activate "Stop" and enter the last value for the sweep parameter.
   The step size is automatically calculated.
   - Or activate "Step" and enter the step size for the value increase or decrease of the sweep parameter.
   The stop value is automatically calculated.
7. Click "OK" to save the settings.

Result: The value of the sweep parameter in the right-clicked row is set to the "Start" value. Additional rows (test steps) with stepwise increased or decreased sweep parameter values are appended under the starting row. The other parameters have the same values for all test steps included in the sweep.

Variation: Sweep over already existing test steps (rows).

1. Press the [Shift] key and click the start and stop row for the parameter sweep. The start and stop row and all rows in between are marked.
2. Right-click to open the "Edit Test Step" context.
   The entry in the "Count" field is fixed to the number of marked columns.
3. Edit the "Sweep Test Steps" parameters.
4. Click "OK."

Result: The selected sweeping parameter sweeps over the marked test steps.

Note: You can select non-contiguous test steps for such a parameter sweep by pressing the [Ctrl] key and clicking each row taking part in the parameter sweep.

6.4 Setting Values by Reference

- Reference to a Test Procedure Parameter
- Reference to a Test Project Parameter
- Reference to a Result Parameter
- Other References

See Table 4-1 for an overview.

6.4.1 Reference to a Test Procedure Parameter

Use case: A parameter in two block functions within the test procedure shall get the value of a common test procedure parameter.

1. In the "Test Procedure Browser" click the "TestProcedure" phase (see **1** in the figure).
   The common parameters for the test procedure are displayed in the "Properties" view.
2. In the "Properties" view, copy the Id of the required common parameter (**2**).

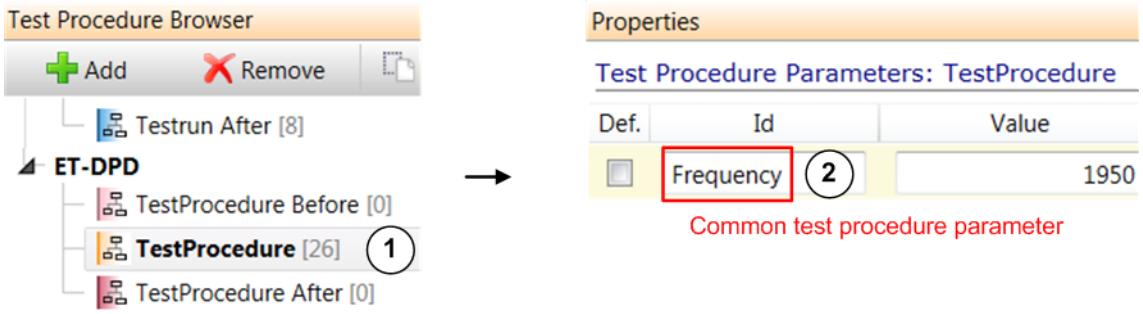


Figure 6-4: Id of common test procedure parameter

3. Click the first desired block in the main view to display the parameters of its selected block function in the "Properties" view (see **3** in the figure).
4. In the "Properties" view, make the desired parameter value editable via check box on the left of the parameter row (**4**).

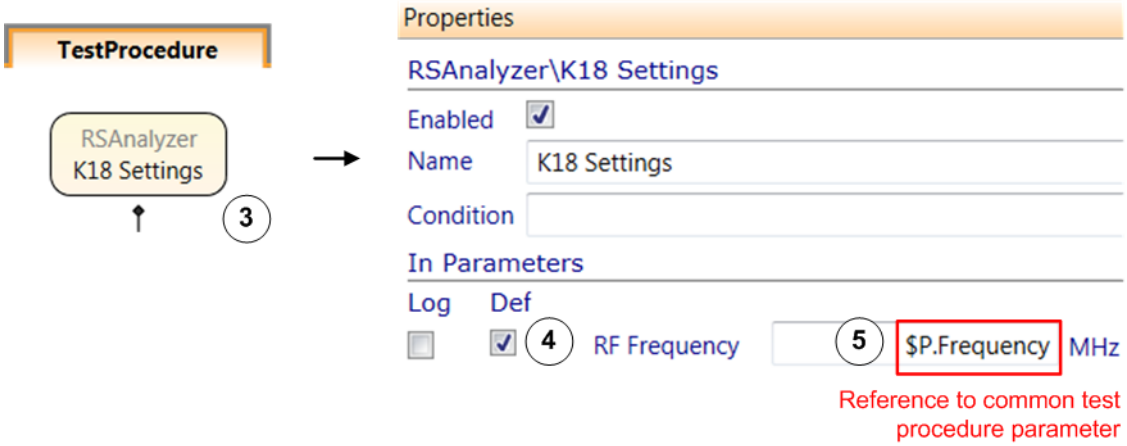


Figure 6-5: Reference to a test procedure parameter

5. Enter the begin of the reference string *$P.* and append (paste) the Id of the common parameter (**5**) or use the "Set Reference" right-click menu.
6. Select the second block function and desired parameter and enter the reference in the same way.

6.4.2 Reference to a Test Project Parameter

Use case: The value of a test project parameter of type "Constant" is used for setting the value of a certain parameter in the test plan table.

For referencing to a test project variable ("Project Param Type" is "Variable"), see Using a Block Function Result as Input for Another Block Function.

1. In the "Test Project Browser" within the "Testplan Editor", click the "Test Project Parameters" item.
   The test project parameters are displayed in the main (middle) view.
2. Right-click the row of the parameter of interest (outside the value fields) and select "Copy reference param" from the context menu.
   The complete reference string $T.<Id> of the test project parameter is copied onto the clipboard.
3. In the "Test Project Browser", click the desired sequence item under the "Test Steps" node.
4. In the test plan table, navigate to the test step and parameter whose value you want to set by reference.
5. Double-click the value field to enter the edit mode, then right-click the value field and select "Paste".

6.4.3 Reference to a Result Parameter

Use case: A measurement result of one test step is used as input value for a following test step.

There are three formats of references to result parameters:

- Full reference: *$R.<Block><BlockFunction><ParameterName><RepNo><TestStepNo><LoopId>*
- Standard reference: *$R.<ParameterName><RepNo><TestStepNo>*
- Short reference: *$R.<ParameterName>*

The full reference is unique. If the standard or short format is used, the reference might not be unambiguous. Any result fulfilling the reference string is taken into account.

The value "<this>" is often appropriate, for example for the repetition number if all repetitions shall be taken into account.

Note that block functions which depend on result references are not started until these references are available.

Variant 1: The reference is manually entered in standard reference format.

1. Make sure you have the following information at hand:
   - Name of the result parameter

- Repetition number (if repetitions are applied)
- Test Step ID where the value is fetched from by reference

You can get the result parameter name from the "Results Viewer" if a previous test result is available.

2. In the "Testplan Editor", select the test step where a parameter shall get its value by reference.
3. In the "Test Step Parameters" tab on the right side, enter the reference string $R.<parameterName><RepNo><TestStep> in the input field of the parameter.

Variant 2: The reference is entered with the help of a test result from a previous test run.

1. In the results table in the "Results Viewer", right-click the result value to be used as input for a test plan parameter. Select "Copy Result Reference".
2. In the "Testplan Editor", paste the copied reference into the input field of the test plan parameter.
   The parameter value is displayed as $R.<BlockName><BlockFunction-Name><parameterName><RepNo><TestStepNo><LoopId>.
3. If necessary, manually adjust details of the reference string. For example, reset <RepNo> to <this>.

Variant 3: The reference is entered with the help of the "Set Reference" dialog.

1. Make sure you have the following information at hand:
   - Name of the result parameter
   - Repetition number (if repetitions are applied)
   - Test Step ID where the value is fetched from by reference
   - Loop ID (if a loop is applied)

   You can get the result parameter name from the "Results Viewer" if a previous test result is available.
2. In the "Testplan Editor", navigate to the test step and parameter whose value you want to set by reference.
3. In the "Test Step Parameters" tab on the right side, right-click the input field of the parameter and select "Set Reference".
   The "Set Reference" dialog is opened.
4. Select "Result Reference" for the "Reference Type".
5. Select the block name and the block function name.

6. Enter/select the other reference information.
7. Click "OK".

   The parameter value is displayed as $R.<BlockName><BlockFunction-Name><parameterName><RepNo><TestStep><LoopId>.

6.4.4 Other References

- Reference to a test procedure variable (whose value can change during test execution): See Using a Block Function Result as Input for Another Block Function.
- Reference to a system configuration parameter: The reference area identifier *$M* is used, see Chapter 6.9, "Using System Configuration Parameters", on page 107.
- Reference to a VISA alias: The reference area identifier *$V* is used, see the QuickStep User Training manual.

6.5 Using a Block Function Result as Input for Another Block Function

The block functions may belong to different blocks.

Situation and realization: The first block function returns a measurement value in a result parameter, for example in measResult1. This parameter is available in the "Out Parameters" section of the block function's "Properties". Then, the measResult1 value (not visible in that section) is stored in a global test project variable. The relevant parameter of the second block function gets its value by $G. reference to the test project variable. Note that a test project variable can get another value during test execution in contrast to a test project parameter (referenced by $T).

It is assumed that the output parameter of block function 1, the receiving parameter of block function 2 and the test project parameter have the same data type and that block function 2 will be executed after block function 1.

Variant 1: Create the global variable beforehand

1. In the "Test Project Browser" at the left side of the "Testplan Editor", add a Test Project Variable.
2. Go to the "Test Procedure Editor" and select block function 1.
3. At the measResult1 parameter (our example) in the "Out Parameters" section of the "Properties" pane, set a reference to the test project variable (*$G. ...*). During test execution, this test project variable will get the value of measResult1.
4. Still in the "Test Procedure Editor", select block function 2.
5. Set the value of the desired block function 2 parameter by reference to the test project variable (*$G. ...* again).

For setting the value by reference you can copy the reference string onto the clipboard (right-click the variable row in the Test Project Parameters table and select "Copy reference param") and paste it into the value field.

Variant 2: Create the global parameter directly via block function 1:

1. Start with block function 1 selected in the "Test Procedure" tab in the "Test Procedure Editor".
2. At the measResult1 out parameter (our example), enter a reference to a test project variable which does not exist yet: *$G. ....*
   A test project variable with that name and required data type is automatically created.
3. Copy the $G reference string via CTRL + C or right-click menu.
4. Still in the "Test Procedure Editor", select block function 2.
5. Paste the $G reference string into the value field of the desired block function 2 parameter.

6.6 Building a Test Procedure

To keep the description simple, only the mandatory test execution phases are considered.

Starting point is the "Test Procedure Editor" view after having created a new test project (hence, all phases are empty).

1. Click the "Blocks & Connectivity" item in the "Test Procedure Browser" on the right side.
   The main area in the middle displays the empty "Blocks & Connectivity" tab.
2. Drag the desired instrument blocks and other required blocks from the "Library" on the left side into the middle area.
   Only these blocks are selectable in any of the test execution phases.
3. If blocks communicate which each other, draw connection lines via mouse: Draw from the out port of the block requesting an activity to the related in port of the block providing information or an activity.
4. In the "Test Procedure Browser", click "Testrun Before" under "MainProcedure".
5. Go to the "Library" (which now offers all blocks present in "Blocks & Connectivity"). For each block to be used in the test procedure and requiring an initialization (at least each instrument block):
   - Expand the block node and drag the "Init" frame into the middle area.
     The block is displayed with the Init function selected.
   - Move the block to the desired position via drag & drop.
   - Enter the values for the block parameters in the "Properties" view.
6. Click the "TestProcedure" node in the "Test Procedure Browser".
   The main area displays the empty "TestProcedure" tab.
7. If you like, press the [F2] key to make the default test procedure name editable and enter an appropriate name for the test procedure.
8. Go to the "Library" again. For each desired block function, expand the block node and drag the desired function item into the middle area.
   The instrument blocks are displayed with the selected function.
   The block function parameters get visible in the "Testplan Editor" where their values are usually set.
9. If you like, you can make a parameter of a block function available in the test plan table by activating the check box for the parameter in the "Properties" view. Thus, the parameter value can be changed from test step to test step. If the predefined checkbox is set, the parameter will have the same value for the whole test procedure and will not be visible in the "Testplan Editor".
10. If a block B shall come into service only after another block A:

- Click block A.
- Position the cursor over a catch element on the block edge.
- Press down the mouse key and move the cursor to a catch element on the edge of block B.
  An arrow line between the two blocks is drawn.

11. Select the "Testrun After" view in the "Test Procedure Browser".
12. Add blocks in the main area with selected Close functions corresponding to the "Testrun Before" settings.

The new test procedure has to be put into operation in the "Testplan Editor":

1. Go to the "Testplan Editor" and click the "Update Test Project" menu item.
   The test procedure gets known for the test plan.
2. Assign the new test procedure to test steps as required.

The procedure is saved with the test project ("File > Save Test Project").

6.7 Evaluating Test Results

Test results are automatically stored in a folder with a time stamp in its name under `.\Results\` of the corresponding project directory.

1. Starting at the "Results Viewer" tab, load the desired test results into the "Results File Browser" (if they are not already there). You have the following possibilities:
   - Initially, the actual content of the default result folder of the project directory is shown.
   - Click "Browse..." in the toolbar. A search window is opened.
     Navigate to the `.\Results\` folder, select the folder with the desired results and click "OK".
   - Drag and drop a folder under `.\Results\` into the "Result File Browser" area.

   The result files are loaded into the "Results File Browser" area.
2. Click the `TestStepsResults.log` file in the "Results File Browser" to have its content loaded into the results table. This file contains the usual result parameters.

3. Inspect the results table.
4. For sorting columns:
   - Single column: Click the header cell of a column for having the result sets (rows) sorted according to the values in that column.
   - Several columns: Press the [Shift] key and click several column headers to get a multiple sorting for those columns.
     The first sorting is done in the column clicked first (usually: "RepNo"). The second sorting (usually: "TestStepNo") is subordinate to the first and affects the result sets with equal values in the first sorting column. These result sets are sorted according to the values of the second column. Further sortings – if you have clicked more column headers – are done accordingly.
5. For getting a diagram representation of results and a statistical evaluation:
   - Click a table cell in the column for the result parameter of interest.
     The diagram displays a line chart for that result parameter over the test steps (by default). The "Histogram & Statistics" view shows the distribution of values and the statistical evaluation.
   - Click a table cell in one column of interest and then the [Ctrl] key + a table cell in another column of interest.
     The diagram displays the line charts for both result parameters.
6. If you want to look for correlations between two result parameters, select a result parameter in the x-axis drop-down box while keeping the y-axis result parameter.

6.8 Building a System Configuration

The system configuration is based on a graphical representation of the test setup in terms of devices and their connections. This procedure describes how to build the graphical representation and how to set the involved parameters.

Starting point: "System Configurator" view with empty default "System 1" configuration

1. Click "System 1" and press the F2 key to make the name editable (or edit the name in the properties window). Then enter an appropriate name.
2. Drag the symbols for the devices of your test setup from the "Device Library" on the left into the main view and arrange the blocks appropriately: Position a

generator and other devices providing output signals on the left side, analyzer and other devices receiving input signals on the right side. A good place for the DUT (using in and out ports) is the middle of the view. Take care that you include also splitters and attenuators if such elements are contained in the physical test setup.

3. Draw the connection lines which are uni-directional between the devices. For each connection between a source (out) and a target (in) device:
   - Position the cursor over the relevant out port (orange circle) of the source device, press and hold down the left mouse key.
   - While holding down the mouse key, move the cursor to the relevant in port. Then release the mouse key.

   An arrow line from the out port to the in port is drawn.
4. Step through the components causing attenuations (e.g. splitters) – click the blocks – and enter the attenuation values in the "Properties" view.
5. Step through the connections – click the connection lines – and enter their attenuation values in the "Properties" view.
6. For defining a new path:
   - Right-click the "Paths" node in the "System Browser" and select "Add Path..." from the context menu.
   - Click the new "Path <Number>" item, press the F2 button and enter an appropriate name for the path.
   - For a connection to be added to the path: Right-click the connection item for that connection under the "Connections" node. Then select "Copy Reference" from the context menu.
   - Right-click again the new created path node and "Paste" the connection item into the path.
   - Repeat these actions for all elements of the path.
7. If you like, save your configuration as `*.sdf` file with the "Export System" button from the toolbar (for example, to reuse it in another project). The configuration is in any case saved with the test project ("File > Save Test Project").

For communicating the system configuration to the test project, go to the "Testplan Editor" and click "Update Test Project" in the toolbar.

6.9 Using System Configuration Parameters

A test step parameter shall get its value by reference from a system configuration parameter. It is assumed that the system configuration is already available.

For using system configuration paths in order to compensate path losses, see Chapter 4.8.2, "Compensation of RF Attenuations via System Configuration Paths", on page 54.

Case A: The desired target parameter is not already available in the system configuration

The desired target parameter is created via the "Mapping Table Editor".

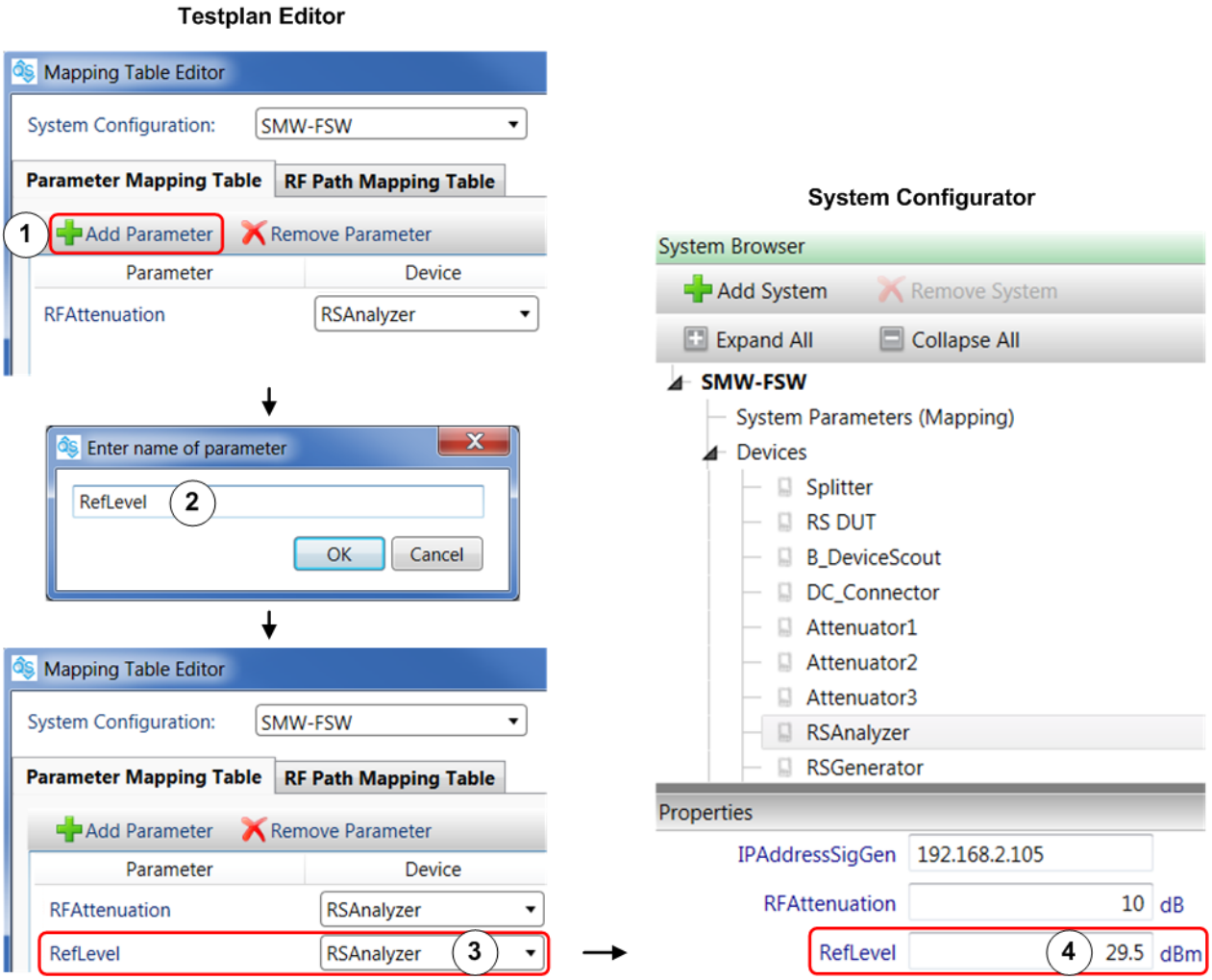


Figure 6-6: Using the Mapping Table Editor

1. In the "Testplan Editor", select "Mapping Table" from the menu.

The "Mapping Table Editor" is opened.

2. Select the desired system configuration.
   This system configuration is used during test execution.
3. Select the "Parameter Mapping Table" tab.
4. Click the "Add Parameter" button and create a new mapping parameter (**1** and **2** in the figure).
5. Optionally, select a device of the system configuration under which this parameter is grouped (**3**).
6. Click the "Save" button.
   The "Mapping Table Editor" is closed.
7. Back in the "Testplan Editor", click "Update Test Project" from the menu.
   The parameter is taken over in the system configuration.
8. Go to the "System Configurator" and identify the parameter:
   - If you have specified a device in step 5, navigate to this device in the "System Browser" and click it.
   - If you skipped step 5, click "System Parameters (Mapping)" in the "System Browser".

   The parameter is displayed in the "Properties" area.
9. Enter a value for the parameter (**4**).
10. Back in the "Testplan Editor", proceed as described in Case B below.

Case B: The desired system configuration parameter is already available

1. In the "Test Step Parameters" tab within the "Testplan Editor", right-click the parameter which shall get its value by reference.
2. Select "Set Reference" from the context menu.
3. In the "Set Reference" dialog, enter select "Mapping Parameter" for the "Reference type".
4. Select the desired system configuration parameter for "Reference to".

When executing the test plan, the current system configuration and mapping are automatically applied.

6.10 Creating a Report Definition

This procedure creates a report definition in RDL format (Report Definition Language) with the ReportDesigner. The report definition is stored in an `.rdlx` file for further use by "RS_Report" block functions. The example report definition in this procedure contains only text boxes with fields which can be filled with result data by QuickStep subreport block functions.

1. In the Test Procedure Editor's toolbar, select "Report Designer".
   The "ReportDesigner" window opens.
2. Select "File > New" from the menu and create a new "Rdl Report".

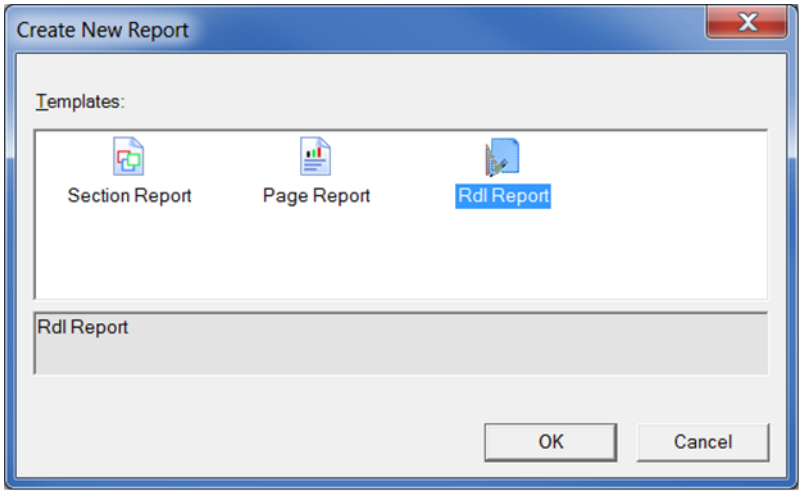


Figure 6-7: New Rdl Report

3. Back in the "ReportDesigner" window, select "Report" in the "Report Explorer" tab on the right and configure its properties in the lower right view.
   The main property is the "Size" defining the height and width of the report area displayed as sheet in the middle view.
4. Drag TextBoxes from the element list on the left into the sheet in the middle view.
5. Arrange the TextBoxes and their sizes according to your needs.
6. Select a TextBox in the "Report Explorer" tab on the right and configure its properties (for example the font) in the lower right view. Repeat this action for all TextBoxes.

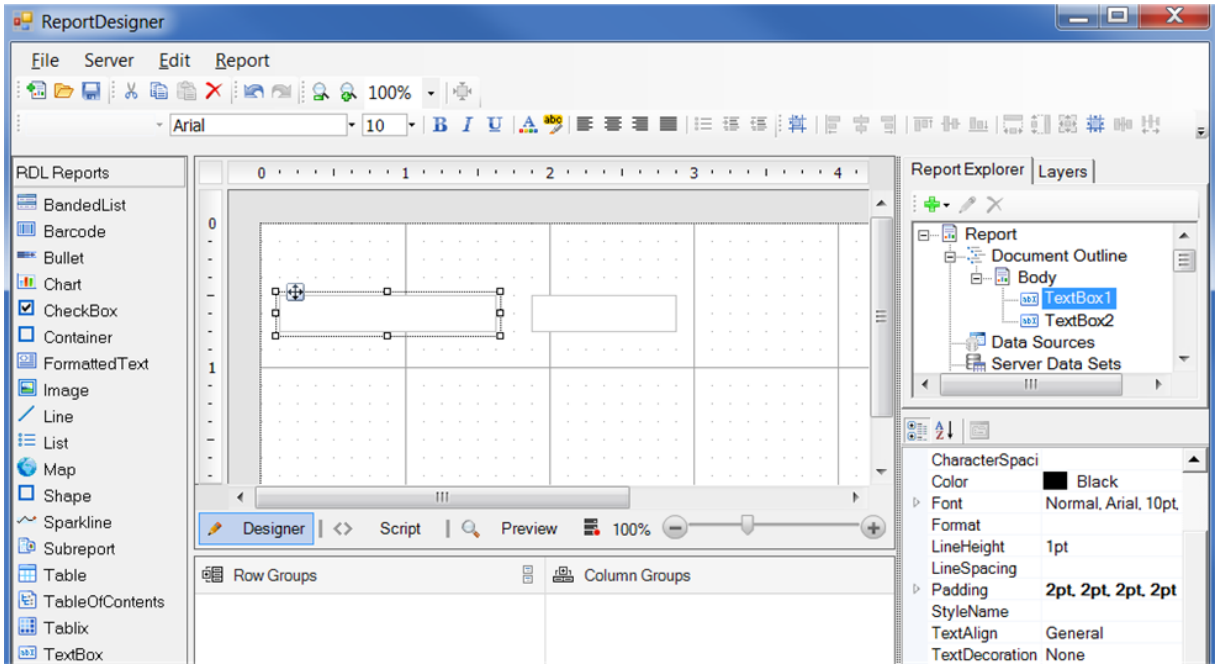


Figure 6-8: TextBoxes and their properties

7. Right-click "Data Sources" in the "Report Explorer" tab and select "Add Data Source".
   The "Report Data Source ..." dialog opens.

8. Change or keep the data source name. Here, it is assumed that the default name is kept: "DataSource1". Select the Type "Dataset Provider" (for manual input) and click "OK" to add the data source.

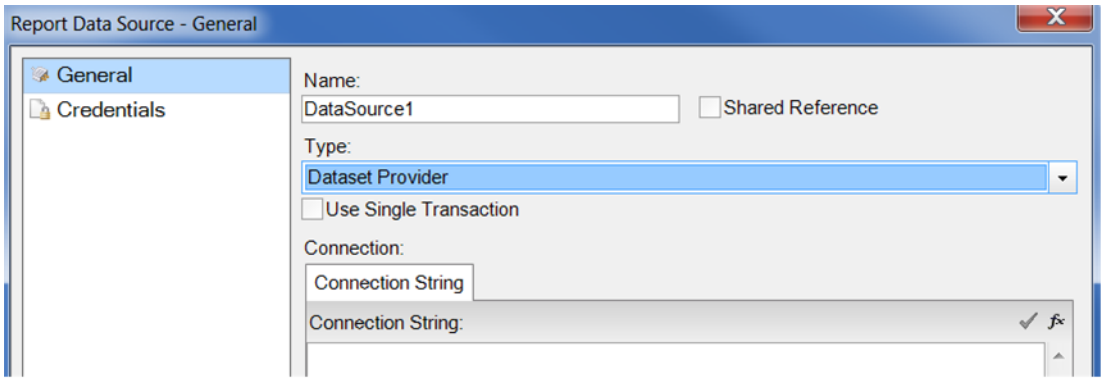


Figure 6-9: Report data source

9. Back in the "Report Explorer" tab, right-click "DataSource1" and select "Add Data Set".
   The "DataSet" dialog opens.

10. Select "Fields" on the left area.

11. In the main area, add Fields, assign or keep Names, and set Values. Then click "OK".

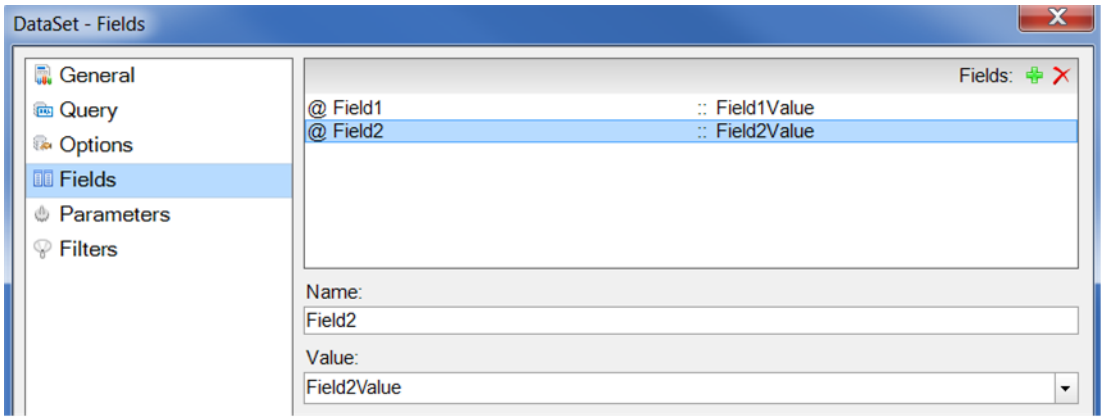


Figure 6-10: DataSet - Fields

The fields appear in the "Report Explorer" under "DataSource1 > DataSet1". The names are used by QuickStep subreport block functions to overwrite the Values with result data.

12. If the report definition will be used for defining the report header, change the names of the fields via "Design > (Name)" in the "Properties" pane. Use only the following names: Title, Subtitle, DeviceUnderTest.

13. In the main area of the ReportDesigner window, select a TextBox. Then, click the configuration icon in the activated TextBox and select one of the previously defined fields. Repeat this for the other TextBoxes.

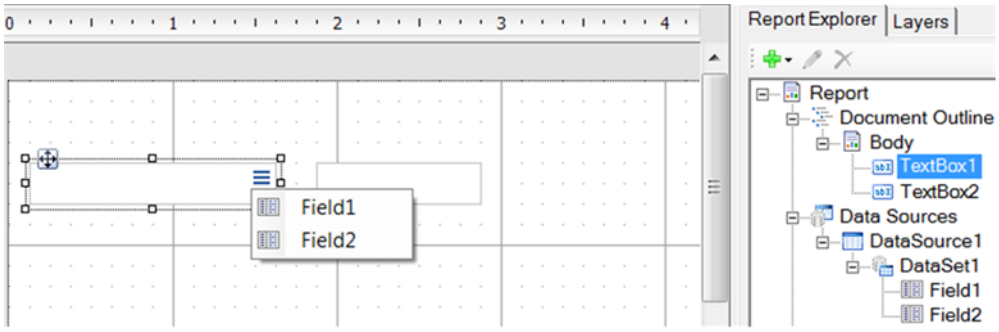


Figure 6-11: Assigning a field to a TextBox

The TextBoxes get the data source fields as content.

14. Double-click the field expression in a TextBox to adjust the expression to "=[Fieldname]".

15. Save the report definition.

Recommended path if used for subreports of different QuickStep projects:
`C:\Users\Public\Documents\Rohde-Schwarz\QuickStep\Common\ActiveReports\Subreport`

6.11 Creating or Modifying a Style Sheet for Reports

A style sheet usable for QuickStep reports is stored in `.rdly-styles` format and is created or edited with the ReportDesigner. It contains a list of styles and their definitions.

1. In the Test Procedure Editor's toolbar, select "Report Designer".
   The "ReportDesigner" window opens.
2. Select "Report > Stylesheet Editor" from the menu.
   The "Stylesheet Editor" window is opened.
3. If you would like to change or add styles in an existing `.rdly-styles` file, select the folder icon in the toolbar and then "Open Stylesheet from File". Then, navigate to the existing style sheet file and open it.
   The styles already defined are displayed in the left pane.

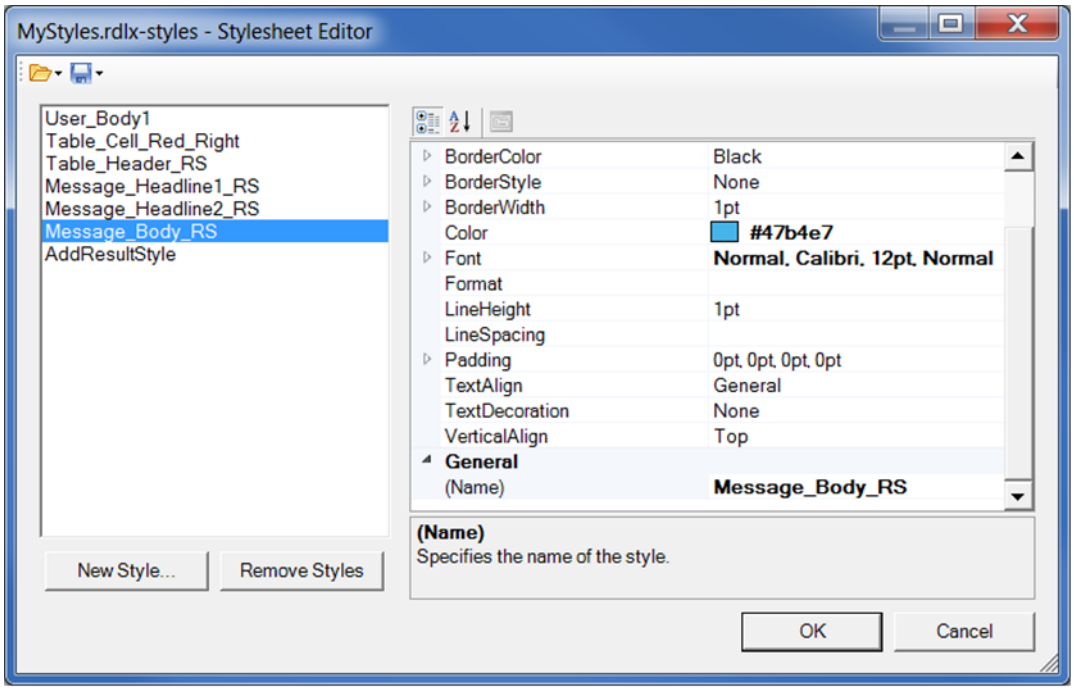


Figure 6-12: Stylesheet Editor

4. Click the "New Style" button to add a new style. Enter a name which is unique in the file and click "OK".

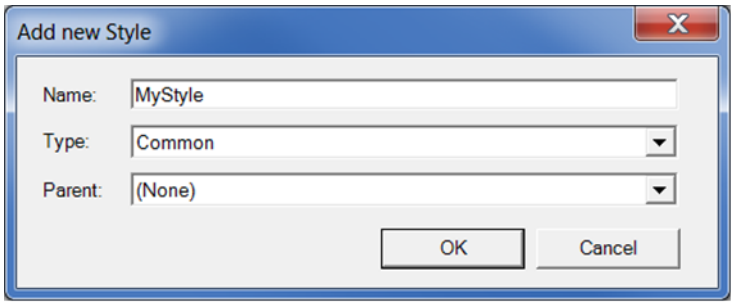


Figure 6-13: Add new Style

The new style is displayed in the left pane of the Stylesheet Editor.

5. Select the new style and edit its properties on the right.

6. Store the style sheet.
Recommended path if used for styles of different QuickStep projects:
`C:\Users\Public\Documents\Rohde-Schwarz\QuickStep\Common\ActiveReports\Styles`

Note: If you store the style sheet under another path, QuickStep might fail to detect it and cannot use the styles.

6.12 Setting Up a Report Including a Subreport

The report set up here contains one subreport. The configuration takes place in the Test Procedure Editor. It is assumed that several DUTs are tested in a row and an own report is required for each DUT. So a report is created in the Test Procedure's "DUT Loop Before" phase.

Prerequisites:

- The required RDL report definition (`.rdlx` file) which a subreport block function connects to is already available, see Creating a Report Definition. In this example, the report definition contains three Fields named Field1, Field2, Field3.
- If a user-defined report header definition is used, it is assumed that its `.rdlx` file is already available.
- If a user-defined style is used, it is assumed that its `.rdly-styles` file is already available. See Creating or Modifying a Style Sheet for Reports.

1. Select "Blocks & Connectivity" in the Test Procedure Browser and drag the "RS_Report" block into the main area.
2. Select the "Testrun Before" phase and set up the "RS_Report > Init" block function.
   This block function offers a live preview of the report during testrun. You can configure the window for the live preview in the "Properties" pane.
3. Select the "DUT Loop Before" phase and set up the "RS_Report > CreateReport" block function.
   This block function creates a report file.
4. Set the properties for the block function.
   - "Type": Select a predefined type or enter path and filename of a user-defined `.rdlx` report header definition. The "Type" determines the appearance of the header part of the report.
   - "DutName", "ReportName", "Title", "Subtitle": Enter text. This text appears in the header part of the report.
5. Select the "Test Procedure" phase and set up a sequence of block functions as shown in the figure.

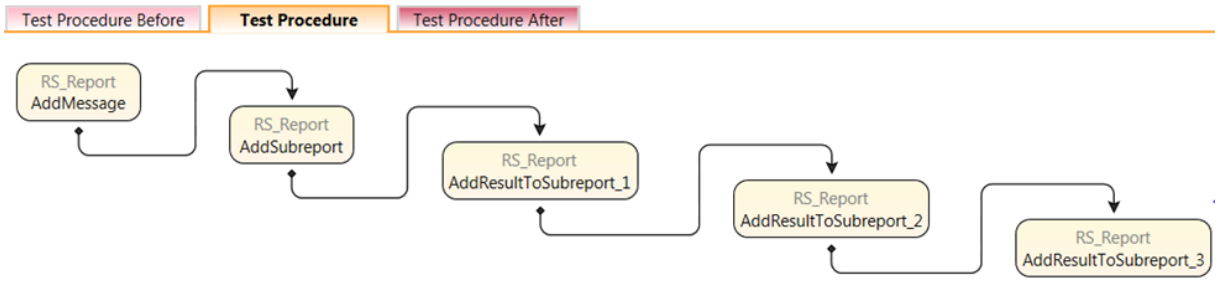


Figure 6-14: Test Procedure for a subreport

6. Configure the block functions:
   - "AddMessage" (optional): Enter a "Text". Select a style from the list at "Style" or enter the name of a style defined in an `.rdly-styles` file.
     This block function adds text in the report's result section according to the selected style.
   - "AddSubreport": Enter an identifying name for "SubreportID". At "SubreportPath", enter path and filename of the `.rdlx` file defining the structure and appearance of the subreport.
     The block function adds a subreport in the report result section.
   - "AddResultToSubreport": Enter the "SubreportID" of the associated subreport. At "FieldValue", enter the name of the desired field defined in

the `.rdlx` file. At "Value", enter the value to be reported in the field (usually, the check box at "Value" is unticked and the value is set in the Testplan Editor).

7. Make sure that the test procedure and test plan are complete regarding the measurement functionality. Update the test plan.

   When running the test, a pdf report is created for each tested DUT. The pdf report is available in the "Results Viewer" under the folder for the DUT results. The result section of each report contains one subreport with three result fields per test step (according to the three "AddResultToSubreport" block functions).

6.13 Using a Shortcut for Test Execution

A test plan can be executed via a shortcut on the desktop. The QuickStep GUI is not involved. This mechanism is helpful for operators who execute pre-defined tests and have no need to work with the QuickStep GUI.

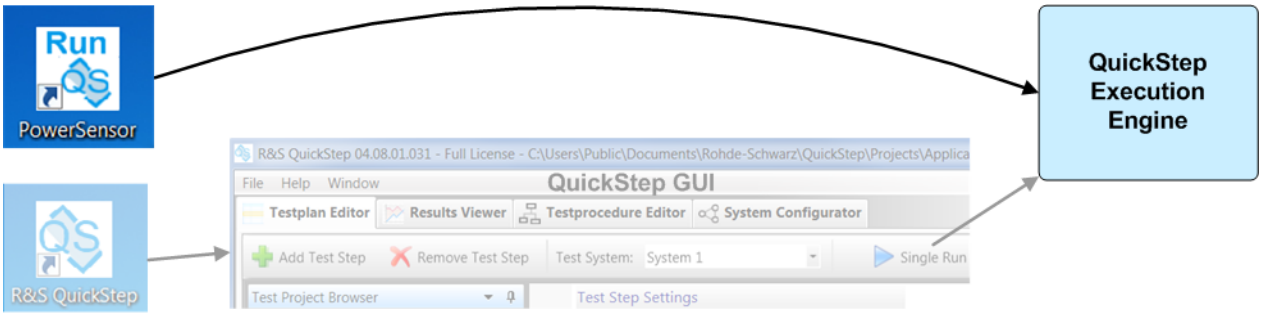


Figure 6-15: Test execution via desktop shortcut

A pre-defined test behind an execution shortcut may contain an application-specific dialog. This dialog opens during test execution allowing the operator to control the test execution.

Creating a test execution shortcut

Starting situation: The test plan (including the connected test procedure) is ready for execution.

► Select "File > Create Testplan Execution Shortcut" from the QuickStep menu.

Result: A test execution icon is added on the desktop with the name of test plan. You can change the name by right-clicking the icon and selecting "Rename".

Executing a test via shortcut icon

► Double-click the test execution icon on the desktop.

Result: The test behind the test execution icon is executed. The QuickStep GUI is not started (if not already running).

6.14 Getting Support on Problems

- Use the contact information for the customer support listed at http:// www.customersupport.rohde-schwarz.com. In case you send a mail, take care to include the product name and version number.
- If QuickStep provides an "Error" dialog, click the "Copy All To Clipboard" button and paste the clipboard content into the mail which you send to the customer support.

"Wrap text" in the "Error" dialog: If activated, long error descriptions in the right dialog area are displayed with line breaks, so the complete content is visible.

7 Block Development

QuickStep can integrate user defined blocks with own functionality. The development of user-defined blocks is usually associated with programming in C++ or C#. Most of the following chapters deal with this kind of block development. Chapter Chapter 7.1.9, "Script Block Functions", on page 150 describes another type of block development where the block functionality is based on script programming.

Block development associated with C++ or C# programming proceeds with the following main stages:

1. Creation of a new block and definition of the basic block properties, particularly block functions and their parameters. Also modification of a user-defined block, for example addition of a new block function
2. Export of the block definitions to programming code including a complete program skeleton for the user-defined block functions
3. Implementation of methods
4. Compilation of the programming code

Afterwards, the DLL and other files for the new block are stored in the QuickStep `UserBlocks\BlockLibrary` folder, and the new block is available for QuickStep (QS). The Block Development Tool (BDT) provides means to carry out the first two steps of the list above via GUI conveniently. The implementation of methods, i.e. the actual programming, and the final compilation is done with Visual Studio or Visual Studio Community.

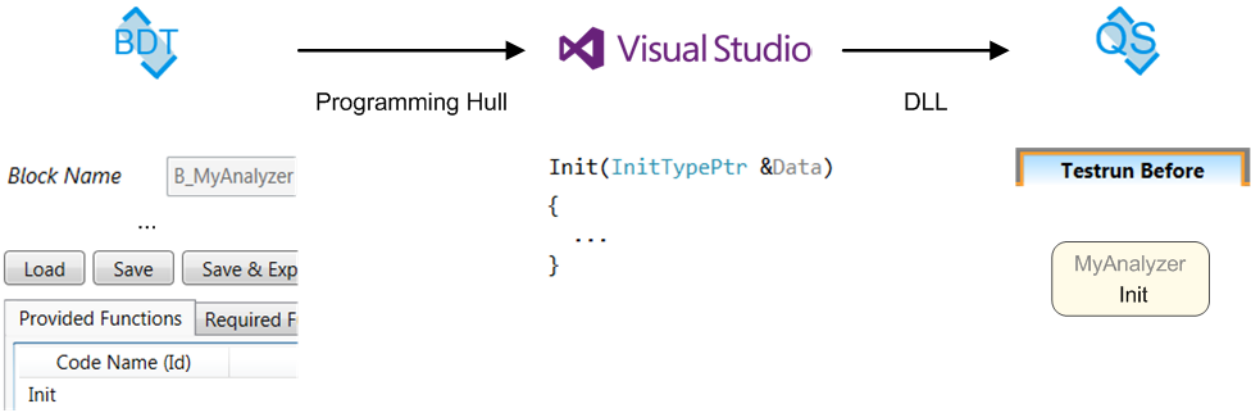


Figure 7-1: Block development overview

Chapter Block Development Concepts describes the applied mechanisms. Chapter Procedures Related to Block Development provides step-by-step descriptions for the main development tasks.

The following descriptions refer to programming with C++ but programming with C# is also supported.

An appropriate IDE is necessary for block development. A Visual Studio installation affords standard C++ and C# block development. Scripting is currently supported with R&S Forum and Mathworks MATLAB. Supported versions are listed in Table 5-1.

7.1 Block Development Concepts

7.1.1 Block Definition

From the user's point of view, the main constituents of a block are its functions – defining what the block can do – and their parameters. The actual block definition contains more components organized as follows:

- **Provided Functions**: Functions realized in the block itself. For example, a function controls a test instrument in a certain way or executes calculations. The functions are implemented in programming code (for example C++, C#) and can include control commands for the test instruments. Input parameters are associated with the functions for determining the concrete calculations, instructions and decisions. A function can also have out (reply) parameters whose values are returned after execution of the function.
- **Required Functions**: Functions (and their parameters) from other blocks used by the block. During test execution the block calls such a function from the block providing that function. Therefore, a communication connection with the block providing that function has to be established. A required function is not included in the list of functions of a block at the GUI (because it is owned by another block and only called).
- **Device Parameters**: Device parameters are means to store certain instrument settings and to avoid unnecessary transmissions of SCPI commands to

the instruments. Therefore, the device parameters are accompanied with special helper functions, for example to check if an instrument setting was changed in the test plan during test execution. For more details, see Chapter 7.1.5, "Device Parameters", on page 133.

- **Block Ports**: The access points for inter-block communication, needed for a block to access functions provided by other blocks. Each port is either an in port or an out port.

Note that the block definition does not require to define the **result parameters** whose measured values are displayed in columns in the results table after test execution. Instead, the result parameters are defined in the programming code (this is a much more flexible method). For defining a parameter as column header for the results table, a `Send…LogResult…(…)` command has to be implemented in the source code, see for example Chapter 7.1.4.1, "Case 1: User-Defined Function for a Software Block", on page 128.

New blocks are created and their properties defined with the R&S Block Development Tool. For adding new functions and parameters to an existing user-defined bloc,k the R&S Block Development Tool is also required.

How To: Chapter 7.2.2, "Creating a New Block and Adding a Function", on page 159

Saving and exporting the new block generates a `B_[BlockName].bdf` file where the functions, parameters and ports are described in XML format. See the following figures showing the definition of the Calculator block as example. The block contains the user-defined Calculate function with the input parameters Num1, Num2, OpMode and the out parameter Result for returning the result of the calculation.

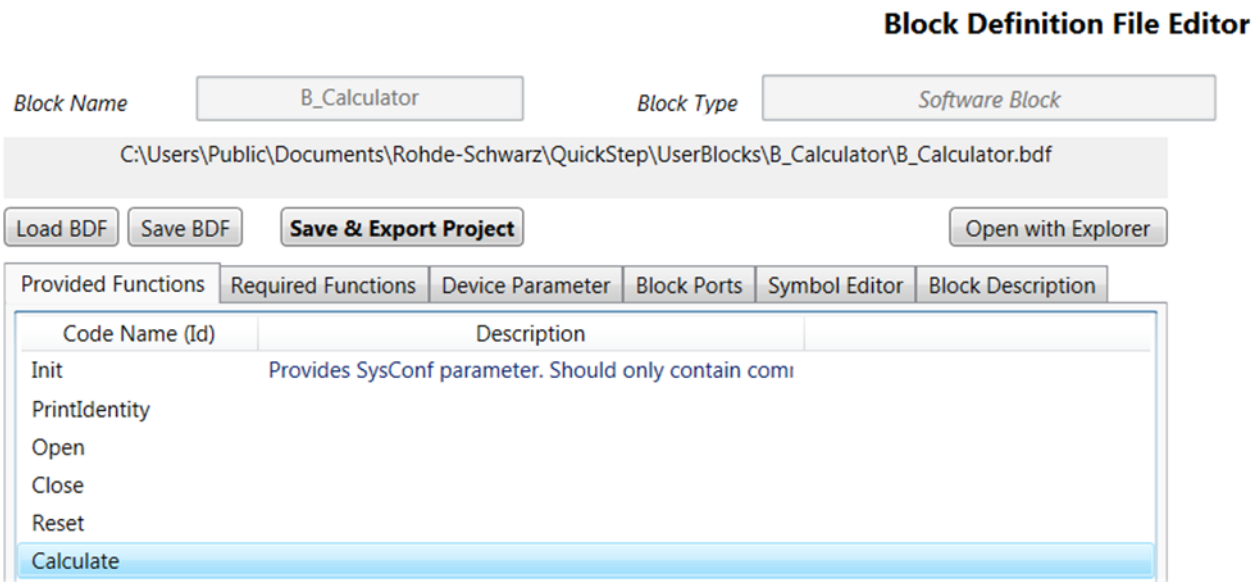


Figure 7-2: Defining the Calculator block and the Calculate function

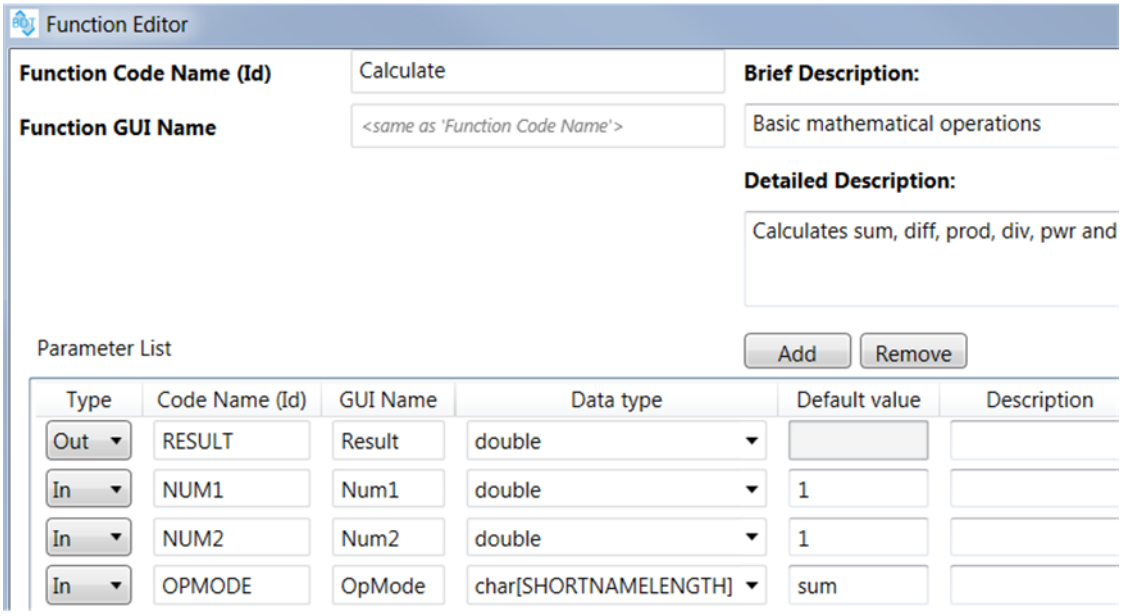


Figure 7-3: Defining the parameters for the Calculate function

A parameter of type "Parameter" is used as input for the function when it is called. A parameter of "Out" type is returned by the called function for further usage in the calling function. "Out" does not mean that the parameter appears as result in the results table after test execution.

Standard functions of a block which are automatically created (for an "Instrument Block with VISA"):

- "Init": Sets system configuration parameters required before executing the test steps. In case of an instrument block, Init is used to set the parameters for the VISA connection to the device.
- "PrintIdentity": Requests the identity information of the instrument via `*IDN?` and writes it into a log file.
- "Open": Can alternatively be used to establish the VISA connection or for specific one-time configurations.
- "Close": Disconnects the block connections, e.g. to the instrument.
- "Reset": Sets the block parameters to their initial (default) values.
- "CheckBlock": Not visible in the Block Development Tool. The function can be used to check the firmware or hardware version of the connected instrument.
- "Catch": Not visible in the Block Development Tool. The function can be used to execute additional commands after an exception occurred, for example to set the device into a known state.
- "SCPI_Read", "SCPI_Write", "SCPI_WriteTillDone", "SCPI_Query": Standard functions to send basic SCPI commands/queries to instrument

If "Enable CheckBlock" has been activated under "TPR Options", then, at the beginning of each test execution, the block functions "Init", "CheckBlock" and "Close" are automatically executed for each block.

7.1.2 Programming Structures

When Saving & Exporting the block definition to code with the Block Development Tool, a complete Visual Studio project is generated. It gets an own folder with the entered name and under the selected path. This folder contains the `B_[BlockName].bdf` block definition file and programming structures within the `*.h` header files and associated `*.cpp` source code files. Project (`*.vcxproj`) and compilation files (e.g. files describing the dll interface) are included to get a complete project. The project can be opened and a solution in Microsoft Visual Studio (Community) can be built.

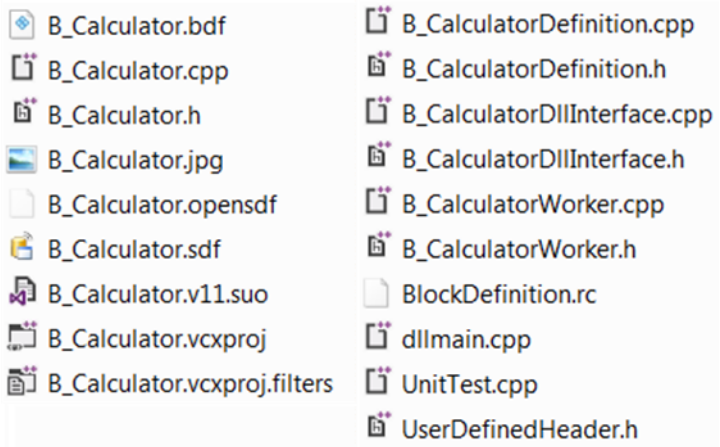


Figure 7-4: Project files (after first "Build") for the user-defined "B_Calculator" block

File dependencies and distribution of content

The following figure shows dependencies between files and how the main programming objects are distributed over the files. The block name in brackets [ ] has been set during block definition.

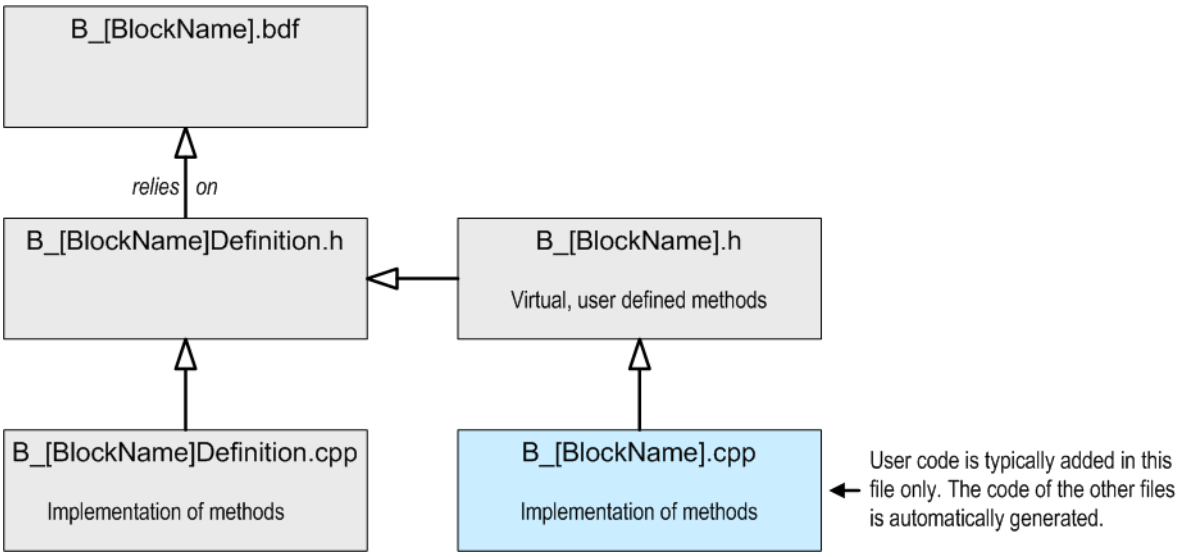


Figure 7-5: Dependencies between code files

The `B_[BlockName]Definition.h` file reflects the `B_[BlockName].bdf` block definition. It defines the block class, data structures and also contains basic methods for addressing the block and for communication with the block. The `B_[BlockName]Definition.cpp` file implements further block methods.

The `B_[BlockName].h` file contains virtual methods for the functions defined in the `B_[BlockName].bdf` block definition. These methods are usually imple-

mented in the `B_[BlockName].cpp` file. There are some standard methods which have automatically been created (for example the Init method) and the user-defined methods (corresponding with the user-defined functions). For all of them the implementation skeletons are already prepared. Usually, you have to adapt some standard methods to your test equipment and – which is the main task by far – have to implement the user-defined methods in `B_[BlockName].cpp`. Look for the methods whose names include the user-defined block function name and insert the functional code there.

The implementation of methods can be outsourced to worker files in order to keep the `B_[BlockName]Definition.cpp` file small, see Chapter 7.1.4.3, "Case 3: Standard Functions with Worker Files", on page 129.

```
class B_Calculator : public B_CalculatorDefinition
{
  public:
    // Constructor
    B_Calculator(std::string Id, std::string BlockName, RS_QuickStepRuntime::RS_LogLevel LogLevel);
    // Destructor
    virtual ~B_Calculator(void);

    //Initialization activities for Block B_Calculator
    using B_CalculatorDefinition::Init;
    virtual ReplyInitTypePtr Init(InitType* &Data);
    // Print identity of block B_Calculator and/or its associated equipment
    using B_CalculatorDefinition::PrintIdentity;
    virtual ReplyPrintIdentityTypePtr PrintIdentity(PrintIdentityType* &Data);
    // Activities to open connections, channels or other communication interfaces
    using B_CalculatorDefinition::Open;
    virtual ReplyOpenTypePtr Open(OpenType* &Data);
    // Activities to close connections, channels or other communication interfaces
    using B_CalculatorDefinition::Close;
    virtual ReplyCloseTypePtr Close(CloseType* &Data);
    // Activities to reset the block and its associated equipment to its default state.
    using B_CalculatorDefinition::Reset;
    virtual ReplyResetTypePtr Reset(ResetType* &Data);
    // Activities to check the block operation
    virtual RS_ActivityBlock::ReplyCheckBlockTypePtr CheckBlock(std::string BlockId);
    // Activities in case of an exception
    virtual RS_ActivityBlock::ReplyCatchTypePtr Catch(std::string BlockId);

    // User block function methods
    using B_CalculatorDefinition::Calculate;
    virtual ReplyCalculateTypePtr Calculate(CalculateType* &Data);
};
```

Figure 7-6: Virtual methods in "B_Calculator.h"

Do not remove automatically created code even if it is not used currently. Otherwise, relations between the project files might get lost and compilation errors might be caused.

The automatically created programming structures are commented in Doxygen style. So, you can directly generate a .chm documentation of your blocks via Doxygen if you have installed Doxygen.

7.1.3 Call and Reply of Block Functions

A set of methods is automatically created for a user-defined block function allowing to call the block function, to get reply messages and for other interactions. The call and interplay of the methods are automatically controlled during test execution by the QuickStep Runtime component. The user has only to provide the functional code within the body of the block function which is called.

How To: Chapter 7.2.3, "Calling a Block Function from Another Block", on page 162

Call and reply make use of "Out" and "In" parameters defined in the Block Development Tool:

- "Out": The function returns a pointer to an array which contains the out parameters as components in the code.
- "In": The function uses input parameters which are available as components of the data array in the code.

Call and reply between blocks – inter-block communication

The currently active block function can call a block function of another block. In this case, the called block function is not explicitly visible in the Test Procedure – only the calling block function is shown. Therefore, the function to be called must have been defined as required function in the calling block and a connection between the blocks must have been established. A call message is sent over that connection which is identified by the involved out port of the calling block. If one or several out parameters are defined in the called block, the calling block gets them over the same connection via a reply message.

The following figures show for an example what preparations are required and how the call and reply works. In the example, the SetReferenceLevel() function of the MyAnalyzer block is executed which calls the GetGenPowerLevel() function of

the MyGenerator block and gets back a reply. The GetGenPowerLevel() function of MyGenerator is declared as required function in MyAnalyzer. Consequently, all necessary code to call the function and to receive the return values is automatically generated and can be used in the MyAnalyzer block.

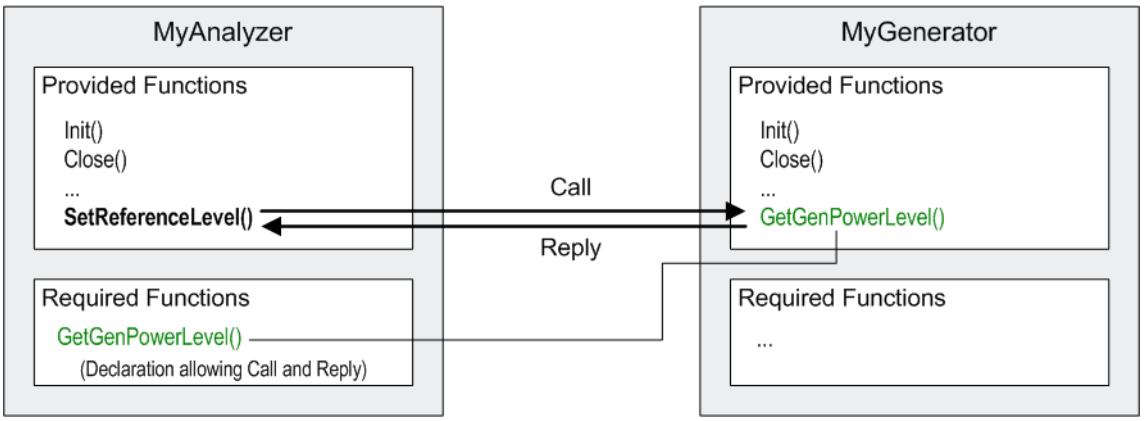


Figure 7-7: Example for communicating blocks

Several applications/tools are involved:

- The provided and required functions are defined in the Block Development Tool.
- The inter-block connection is set up in the "Blocks & Connectivity" tab within the "Test Procedure Editor" in QuickStep.
- The actual call of the MyGenerator's function is implemented in the C++ code via Visual Studio.

Colored frames and font colors in the figures indicate correspondencies between data in the different applications/tools.

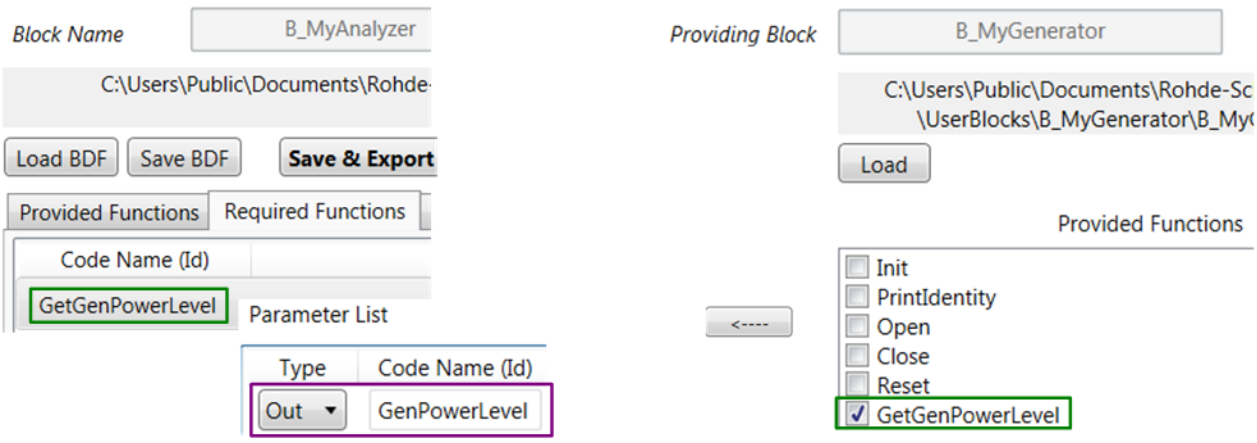


Figure 7-8: Block Development Tool: Declaring the required function

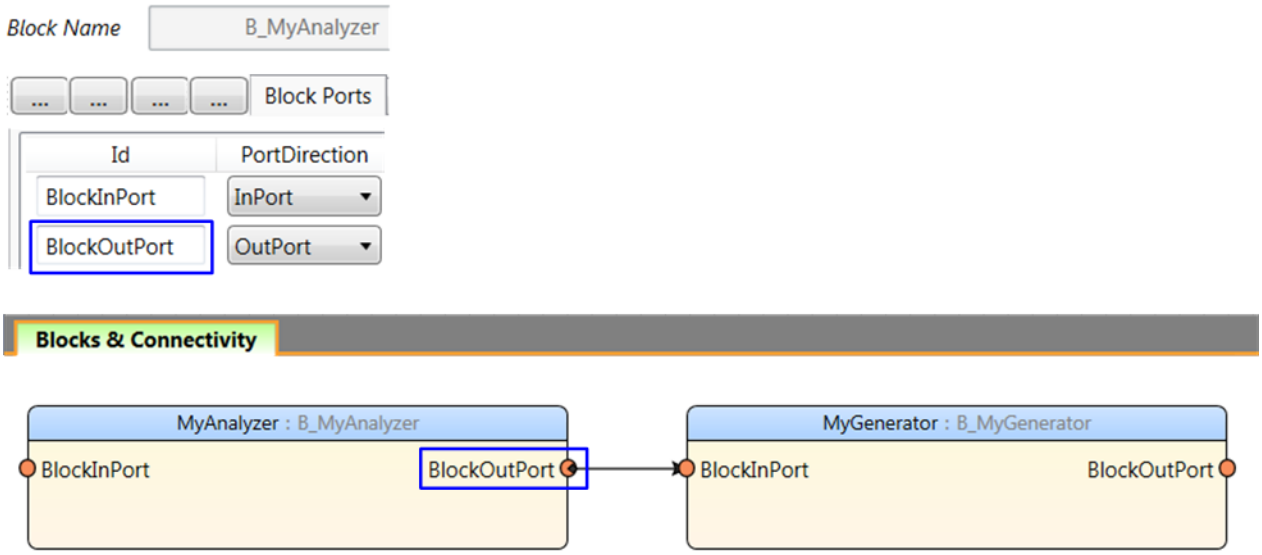


Figure 7-9: QuickStep: Drawing the connection for inter-block communication

The arrow at the connection indicates the direction in which the request is sent – from the originating block to the receiving block. The following notion might be useful: The function call is put inside the MyAnalyzer's BlockOutPort port. As this port is connected to the MyGenerator's BlockInPort port, the MyGenerator block receives this message.

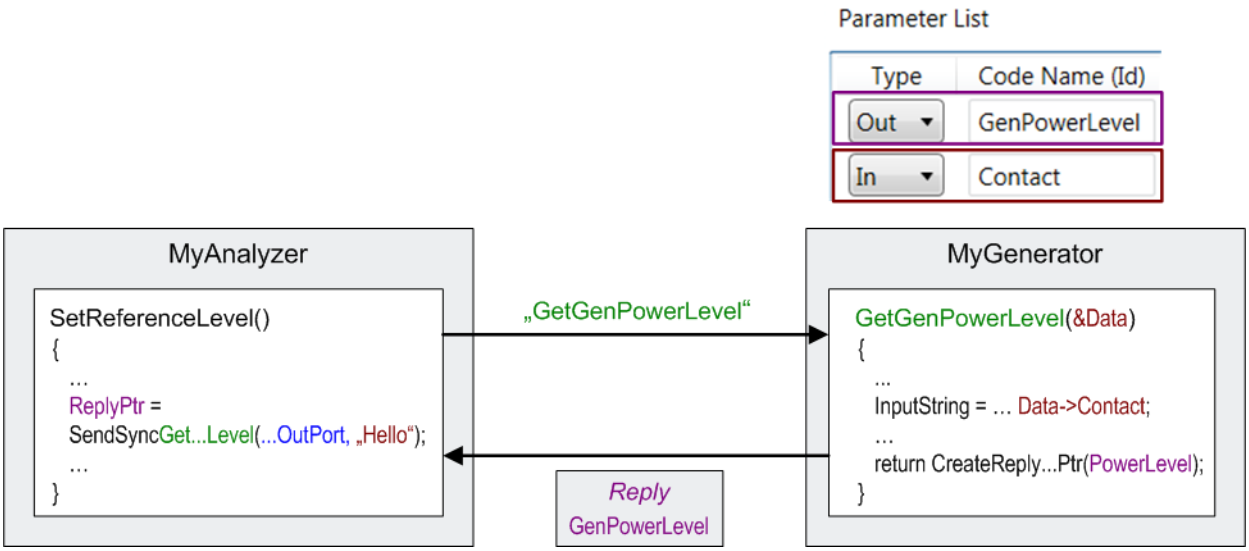


Figure 7-10: Block function call via SendSync and with input parameter

Synchronous or asynchronous call and reply between blocks

There are methods for synchronous and asynchronous calls. Reply refers to the out parameters for the called function as defined in the Block Development Tool.

- Synchronous call (with or without reply): The QuickStep Runtime gets a response message to the call and awaits this message before proceeding to the next execution step.
- Asynchronous call without reply: QuickStep Runtime does not wait for the response message and proceeds to the next execution step. If the required block function has a defined a out value, this value is not returned. Instead, it is deleted because it cannot be used.
- Asynchronous call with reply: QuickStep Runtime does not wait for the response message and, if available, the reply. The response message and the reply can be checked later on during program execution within the block function with WaitForAsyncReply(). When the reply is explicitly requested by the WaitForAsyncReply() command, QuickStep awaits this reply before proceeding to the next execution step. The port used for the asynchronous call with reply is blocked until the reply has been received. In the meantime, no further asynchronous or synchronous call can be sent out on the same port (but it is still possible to send other calls over another connection). It makes sense to wait for the response message if the executed required function shall be finished for successive commands of WaitForAsyncReply().

Execution of many block functions

If a block function calls many other block functions, the incoming messages are queued in the associated block instance, that is in the ingoing queue of its related in-port. The block instance executes the queue one by one. If multiple in-queues exist, all the queues are polled and executed. In case of asynchronous calls, this may lead to the following behavior: Messages arriving at a later stage but stored within an empty queue may be executed first.

The three communication types synchronous (SYNC), asynchronous (ASYNC) or asynchronous with reply (ASYNC_REPLY) are internally separated, but SYNC and ASYNC_REPLY are sent through the same connection. Therefore, they share the same in-queue.

7.1.4 Code Implementation

The implementation of the user-defined functions for a block is done in the `B_[BlockName].cpp` file – or in the `B_[BlockName]Worker.cpp` file if that file is used (see below). The environmental code has already and automatically been created during code export and solution build.

Finally, for compiling the source code files, the block project has to be built in Visual Studio (Community). The DLL and other files for the new block are stored in the QuickStep `BlockLibrary` folder, and the new block is available for QuickStep (QS).

7.1.4.1 Case 1: User-Defined Function for a Software Block

"Software" block indicates that the block is not controlling an instrument. So, no communication over VISA etc. has to be provided.

Example: The Calculator block, the Calculate() function, its input parameters NUM1, NUM2, OPMODE and its out parameter RESULT have been defined with the Block Development Tool. The Calculate() function calculates the sum or difference of the two parameters NUM1 and NUM2. When the test is executed, the calculated result values per test step are logged in the results file and appear in the "Result" column of the result table.

For implementation, only the functional code within the Calculate() function is added in the `B_Calculator.cpp` file.

```
ReplyCalculateTypePtr B_Calculator::Calculate(CalculateType* &Data)
{
    // Add your code here
    double result;
    // Create an output into the LogViewer of the GUI which states the selected operation
    std::string str(Data->OPMODE);
    SendLogConsole(RS_QuickStepRuntime::RS_LogLevel::NORMAL,"Operation: %s",RS_B_CommonPublic::ToCharPtr(str));
    // Execute the operation
    if (RS_B_CommonPublic::StringsAreEqual(str,"sum"))
    {
        result = Data->NUM1 + Data->NUM2;
    }
    else if (RS_B_CommonPublic::StringsAreEqual(str,"diff"))
    {
        result = Data->NUM1 - Data->NUM2;
    }
    else
    {
        SendLogConsole(RS_QuickStepRuntime::RS_LogLevel::WARN,"No valid operator defined!");
        result = 0;
    }
    // Create an output into the result file
    SendAsyncLogResultDouble(RS_QuickStepRuntime::RS_ResultFileType::RESULT,"Result","",4,result);
    // Be sure to fill the reply type ptr appropriately
    return CreateReplyCalculateTypePtr(result);
}
```

Figure 7-11: Implementation of the user-defined Calculate() function

Notes:

- A parameter `parameterName` (defined via Block Development Tool) is accessed via `Data->parameterName`.
- The SendAsyncLogResultDouble(..., "Result:", result) function defines that the result value is written into the result log file. The result values per test step are displayed in the "Result" column in the results table.
- The SendLogConsole() function determines that status information is reported in the "Log Viewer" of the QuickStep GUI.

7.1.4.2 Case 2: Standard Functions for an Instrument Block

An instrument block is a block communicating with a test instrument.

Some standard functions are automatically created during export of the block definition to code and building of a solution with Visual Studio. The Init(), PrintIdentity(), Close( ) functions and the basic block functions to directly send SCPi commands take part in the communication of an instrument block with its associated instrument.

Example: The MyDcSupply instrument block communicates with a DC supply device over VISA and GPIB. These block functions are automatically created. No additional coding is required for those functions.

7.1.4.3 Case 3: Standard Functions with Worker Files

Example: The automatically created PrintIdentity() function for instrument blocks is implemented via worker files.

Worker files

The worker files are automatically generated for each block, but it is the decision of the developer whether to use them or not.

The worker files collect outsourced code in order to keep the main implementation file `B_[BlockName].cpp` lean and clear. It is recommended to put routine, background, 2nd and lower-level functionality and extensive calculations into the worker files. `[BlockName]Worker.h` defines the worker class. The `B_[BlockName]Worker.cpp` file implements the worker methods. An instance of the worker class is created in the constructor of the block:
`Worker = new B_[BlockName]Worker( this );` (automatically gener-

ated). This line should not be removed even if the worker is not used! A method in the `B_[BlockName]Worker.cpp` file is called from the `B_[BlockName].cpp` in this way:

```
ReturnValue = Worker->BlockFunctionName(FunctionParameters);
```

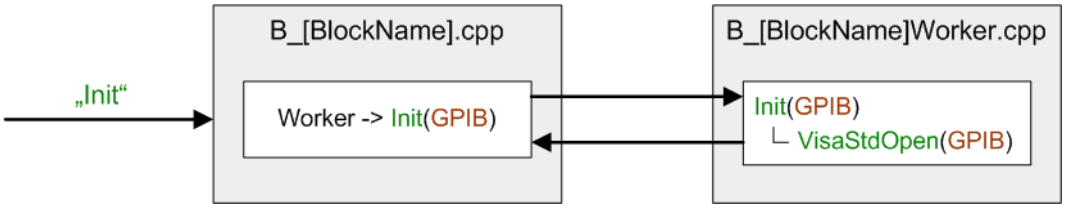


Figure 7-12: Outsourcing code in ...Worker.cpp (example)

Implementation

The implementation affects three files:

- `B_MyDcSupplyWorker.h` where the PrintIdentity() function is declared.

```
class B_MyDcSupplyWorker // add appropriate inheritance
{
public:
    // Constructor of block worker
    B_MyDcSupplyWorker( B_MyDcSupplyDefinition* Block );
    // Destructor of block worker
    virtual ~B_MyDcSupplyWorker(void);

    // Implement your methods here, e.g.
    // Example Open
    int Open( );
    // Example Close
    int Close( );

    // PrintIdentity function declaration
    std::string PrintIdentity();
```

Figure 7-13: PrintIdentity declaration in ...Worker.h

- `B_MyDcSupplyWorker.cpp` where the PrintIdentity() function is defined. The prefix “Block->” is required in the `B_MyDcSupplyWorker.cpp` file to access the functions linked to the block (e.g. VisaStdOpen()). This prefix is not required when the functions are directly called in the `B_MyDcSupply.cpp` file.

```
std::string B_MyDcSupplyWorker::PrintIdentity()
{
char ReadStr[DEFAULTBUFSIZE];
// Send the *IDN? command over Visa
Block->VisaStdWrite("*IDN?");
Block->VisaStdRead(ReadStr);
return ReadStr;
}
```

Figure 7-14: PrintIdentity implementation in ...Worker.cpp

- `B_MyDcSupply.cpp` with the B_MyDcSupply's PrintIdentity(...) function from where the worker's PrintIdentity() function is called.

```
ReplyPrintIdentityTypePtr B_MyDcSupply::PrintIdentity(PrintIdentityType* &Data)
{
    // call the Worker function and retrieve the result string
    std::string IdnString = Worker->PrintIdentity();

    // print the information        (e.g. by sending it to the logging console)
    SendLogConsole( RS_QuickStepRuntime::RS_LogLevel::NORMAL, "%s", IdnString.c_str());

    return CreateReplyPrintIdentityTypePtr();
}
```

Figure 7-15: PrintIdentity call in ...MyDcSupply.cpp

7.1.4.4 Case 4: User-Defined Functions for an Instrument Block

Example: The functions SetVoltage with parameter Voltage, SetCurrent with parameter Current and MeasureCurrent have been defined for the MyDcSupply block and are implemented.

The implementation steps are the same as in the previous chapter:

- The functions are declared in `B_MyDcSupplyWorker.h`.
- The functions are implemented in `B_MyDcSupplyWorker.cpp`.
- The functions are called in `B_MyDcSupply.cpp`.

The figures show the code implementation, particularly how SCPI commands are used in the code.

```
class B_MyDcSupplyWorker // add appropriate inheritance
{
public:
    // Constructor of block worker
    B_MyDcSupplyWorker( B_MyDcSupplyDefinition* Block );
    // Destructor of block worker
    virtual ~B_MyDcSupplyWorker(void);

    // Implement your methods here, e.g.
    // Example Open
    int Open( );
    // Example Close
    int Close( );

    std::string PrintIdentity();

    void SetVoltage(double Voltage);
    void SetCurrent(double Current);
    double ReadCurrent();
```

Figure 7-16: Declaration of functions in ...Worker.h

```
void B_MyDcSupplyWorker::SetVoltage(double Voltage)
{
char WriteStr[DEFAULTBUFSIZE];
sprintf_s(WriteStr, DEFAULTBUFSIZE, ":SOUR:VOLT %lf", Voltage);
Block->VisaStdWrite(WriteStr);
}

void B_MyDcSupplyWorker::SetCurrent(double Current)
{
char WriteStr[DEFAULTBUFSIZE];
sprintf_s(WriteStr, DEFAULTBUFSIZE, ":SOUR:CURR %lf", Current);
Block->VisaStdWrite(WriteStr);
}

double B_MyDcSupplyWorker::ReadCurrent()
{
char Readstr[DEFAULTBUFSIZE];
double Result;
Block->VisaStdQuery(":MEAS:CURR?", Readstr);
Result = atof(Readstr);
return (Result);
}
```

Figure 7-17: Implementation of functions in ...Worker.cpp

```
ReplyMeasureCurrentTypePtr B_MyDcSupply::MeasureCurrent(MeasureCurrentType* &Data)
{
    // Add your code here
    double MeasuredCurrent = Worker->ReadCurrent();

    // Write the measured value into the result file
    SendAsyncLogResultDouble(RS_QuickStepRuntime::RS_ResultFileType::RESULT,"Current","A",4,MeasuredCurrent);

    // Be sure to fill the reply type ptr appropriately
    return CreateReplyMeasureCurrentTypePtr(MeasuredCurrent);
}
```

Figure 7-18: MeasuredCurrent call in ...MyDcSupply.cpp

7.1.4.5 Case 5: Call of a Required Function

Example: The SetReferenceLevel() function calls the GetGenPowerLevel() function and gets the returned data of that function. It is assumed that all preparations (particularly declaring GetGenPowerLevel() as required function in the calling block) have been done.

```
ReplySetReferenceLevelTypePtr B_MyAnalyzer::SetReferenceLevel(SetReferenceLevelTypePtr &Data)
{

  // Add your code here

  ...
  ReplyGetGenPowerLevelTypePtr Reply =  SendSyncGetGenPowerLevel(B_MyAnalyzerPorts.BlockOutPort);

  SendLogConsole(RS_QuickStepRuntime::RS_LogLevel::NORMAL, "Received GenPowerLevel: %f", Reply->GenPowerLevel);

  // Be sure to fill the reply type ptr appropriately
  return CreateReplySetReferenceLevelTypePtr();
}
```

Figure 7-19: Synchronous call with reply in ...MyAnalyzer.cpp

7.1.5 Device Parameters

If a test plan contains parameters for instrument settings (for example the `Current` parameter of the `SetCurrent` function for a DC supply block), their values are set for each test step. The instrument gets the values via SCPI commands, also for each test step. It is typical that many parameters only change from time to time and not in each step.

General					SetCurrent	SetVoltage
No	Id	Enable	Breakpoint	Test Procedure	Current	Voltage
1	1	☑	☐	Test Procedure 1	0.5	2.0
2	2	☑	☐	Test Procedure 1	0.5	2.2
3	3	☑	☐	Test Procedure 1	0.5	2.4

Figure 7-20: Instrument setting parameters in a test plan (example)

Sending a SCPI command for instrument setting should be avoided if the related parameter is not changed (the instrument is already set correctly). In this way, the overall test execution time is kept as short as possible. Avoiding unnecessary SCPI commands is realized with device parameters which provide special helper functions for that purpose. Basically, a device parameter can be used to store a specific status of the instrument or the block. The helper functions are used to decide if a SCPI command transmission or another activity is necessary or not.

How To: Chapter 7.2.4, "Using Device Parameters", on page 165

The following figure shows how device parameters are applied to avoid unnecessary transmissions of SCPI commands.

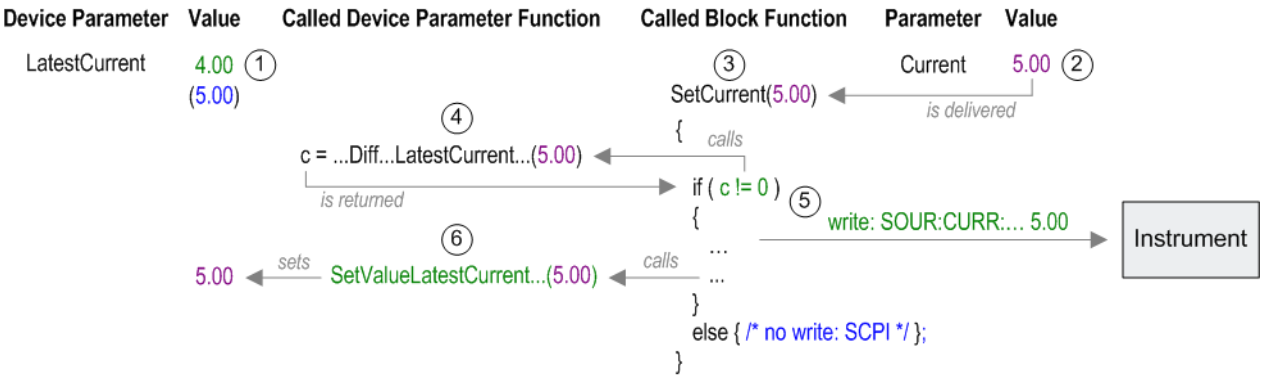


Figure 7-21: Device parameter mechanism

1. Starting situation: The latest value of the `Current` parameter has been stored in the `LatestCurrent` device parameter during test execution.
2. The `Current` parameter gets a new value from the test plan table when the test execution proceeds to the next test step. This new value can be equal to that of the previous test step or not.
3. The `SetCurrent(...)` function which contains the `Current` parameter is called during test procedure execution for the test step.

4. The `SetCurrent` function calls the `IsDirtyOrDiff...LatestCurrent...(...)` helper function and gets the reply whether the new value of `Current` differs from the value of `LatestCurrent`.
5. If the `Current` and `LatestCurrent` values differ, a SCPI command is sent to the instrument with the new value of `Current`.
6. In this case, the `LatestCurrent` device parameter value is updated via the `SetValueLatestCurrent...(...)` function.

The `IsDirtyOrDiff...LatestCurrent...()` and `SetValue...()` methods and some other device parameter methods are automatically created when the block definition containing the device parameter is exported to code and the block solution is built. These methods are called manually in the related block function code, see the figure.

```
SetCurrent(Current)
{
  // Function parameter: Current     Device parameter: LatestCurrent
  if (Block->IsDirtyOrDiffOrForcedLatestCurrentSetting(Current))
  {   ...
      // send SCPI command
      RS_QuickStepRuntime::Snprintf(WriteStr, DEFAULTBUFSIZE, ":SOUR:CURR %lf", Current);
      Block->VisaStdWrite(WriteStr);
      // set Device Parameter to new value
      Block->SetValueLatestCurrentSetting(Current);
      ...
}
```

Figure 7-22: Device parameter: SetValue...()

For a detailed list of available functions dealing with device parameters, see the Developer Training manual.

- At the beginning of a test execution, all device parameters are set to dirty by default (no skip of SCPI command transmissions at start of a test run).
- The device parameter mechanism is not applicable for Write SCPI / Read SCPI / Query SCPI block functions (because QuickStep does not know which parameter is handled within such a SCPI command).

7.1.6 Extension Blocks

A block provided with QuickStep cannot be extended directly with additional functionality because its source code is not delivered. Instead, an additional, user-defined block ("extension block") can be created carrying the required additional functions. Adding an extension block to an existing software block does not cause any interferences. For instrument blocks which control an instrument via a VISA connection further attention is required. Here, the extension block uses the same VISA connection as the provided block; the VISA resource is shared. The sharing concept requires an appropriate internal handling of the Init and Close functions by which a VISA connection is established and closed. The concept is described below.

Shared VISA resource

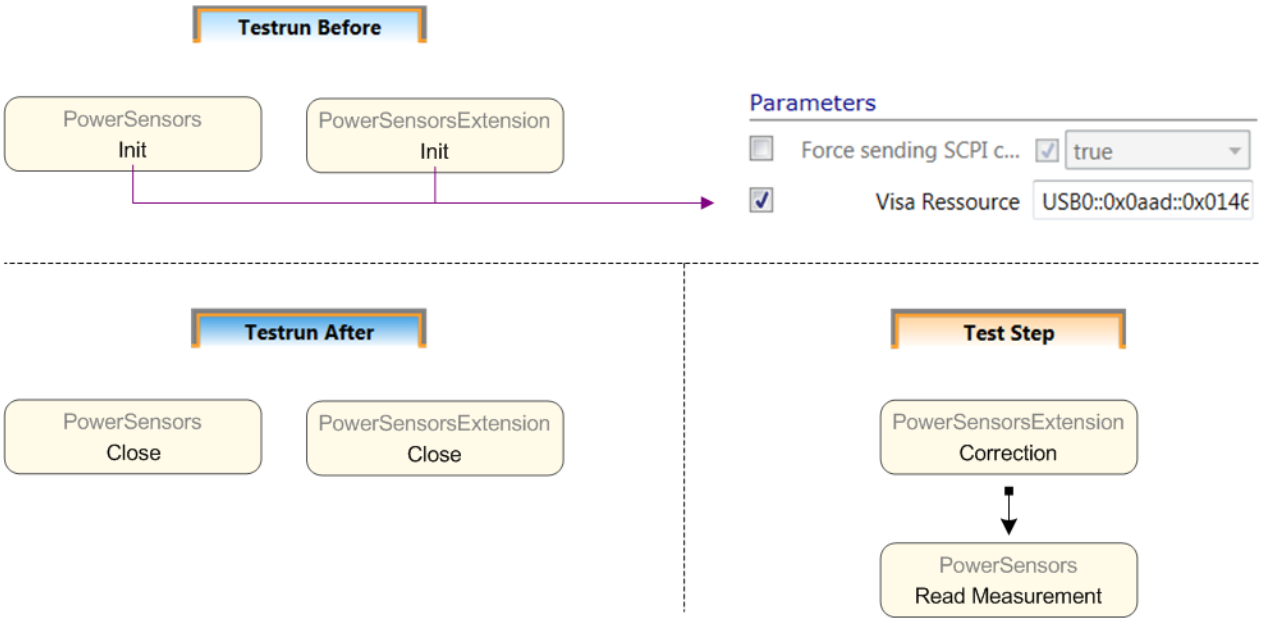


Figure 7-23: Provided block and its extension block in a test procedure

- The extension block contains an Init and a Close function as the provided block (as all instrument blocks).
- It does not matter which Init function is executed first, that of the provided block or that of the extension block.
- QuickStep counts the number of blocks using one VISA connection to an instrument. The actions for Init and Close depend on the number of blocks currently using the VISA connection.
- Init:

  - The first Init function to be executed opens the VISA connection with the specified resource string and returns the unique connection ID.
  - If another Init is executed with the same resource string, it returns the same connection ID as the first Init function.
- Close:
  - The first call of Close does not close the VISA connection if more than one block currently use the connection.
  - The Close call of the last connected block finally closes the connection.

Shared device parameter

Special attention is required when extending a provided block which has device parameters: For example, if the extension block modifies instrument settings without adjusting the corresponding device parameter, the original block might not execute required SCPI commands. It still assumes that the instrument is correctly set. This malfunction is avoided by sharing the device parameters for settings which are used or set by both blocks. This is supported by the QuickStep framework. For additional details, see the corresponding Developer Training Manual.

7.1.7 Block Functions for Direct DLL Call

Preliminary: Block functions for QuickStep

Development of a user-defined block leads to a block DLL which is usually employed by QuickStep: First, the block with a selected block function is added in a test procedure; second, QuickStep calls the DLL to carry out the block function during test execution. QuickStep makes use of its runtime environment and its knowledge of the block functions. That is why the block function may use functions contained in the QuickStep runtime package. These functions mainly provide tracing, logging, VISA connection and timer functionality.

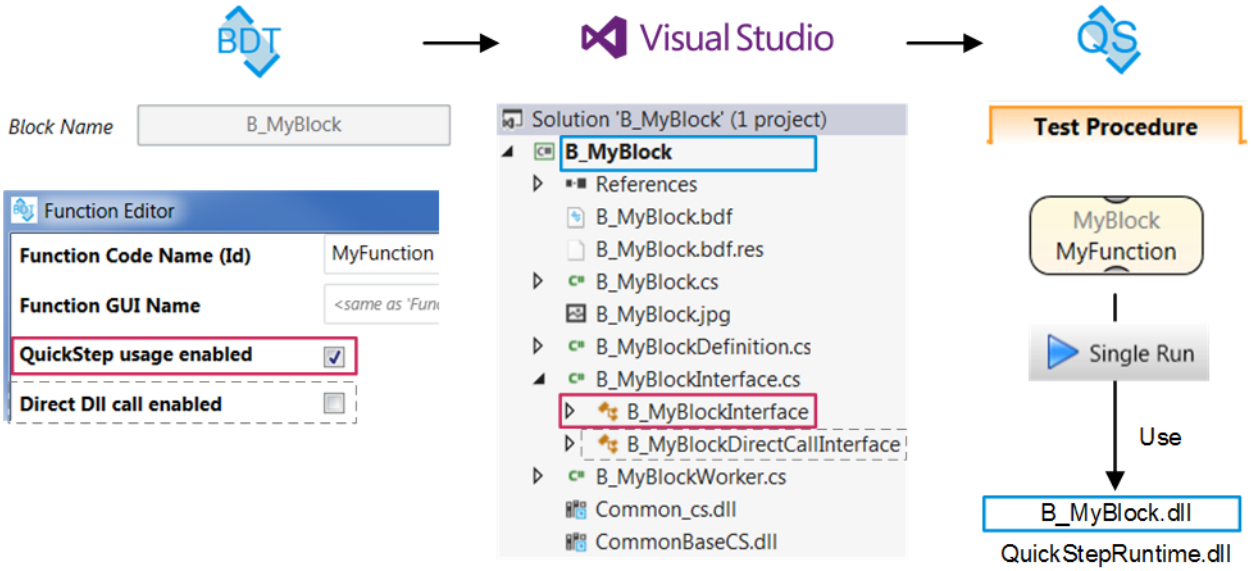


Figure 7-24: Development of block function to be used in QuickStep

Block functions for direct DLL call in user application

QuickStep allows to develop block DLLs with block functions that can directly be called by a customer application and without using QuickStep during call and execution. Therefore, QuickStep exports the block functions (the DLL interface provides the entry points for the functions). For compensating the QuickStep runtime DLL, which is only usable with QuickStep running, the CustomerRuntimeBase DLL is integrated in the compiled solution as embedded resource. It provides QuickStep runtime functions which the block functions or the customer application can use.

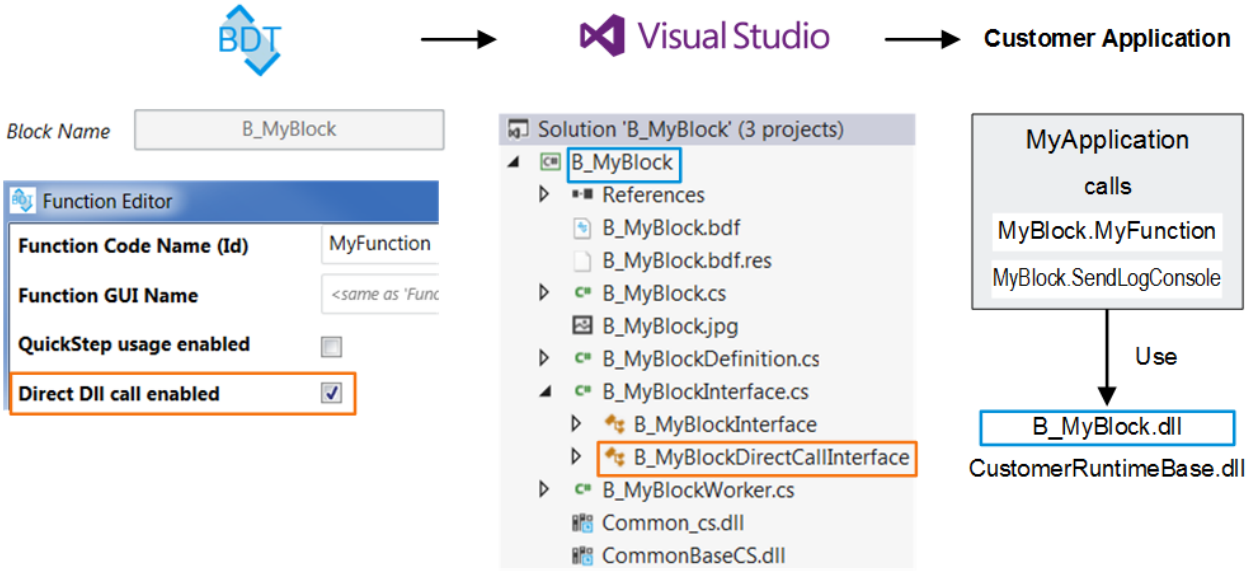


Figure 7-25: Development of a block function used without QuickStep

The direct DLL call property of a block function is already defined in the Block Development Tool (only available if C# has been selected for the programming language). The project information created with "Save & Export Project" instructs Visual Studio to include the function in the DLL interface as exported function when the project is compiled and the solution is built.

The following figure shows an example of a customer program for calling "MyFunction", defined in the "B_MyBlock" block, from the "B_MyBlock.dll". "MyFunction" can use the functions of the CustomerRuntimeBase (embedded resource). References of the application program to B_MyBlock, Common_cs and CommonBaseCS are required. QuickStep is not needed when the MyApplication program is executed.

```
using System;

using RS_B_MyBlock;  ①

namespace MyBlockApplication
{
    class Program
    {
        static void Main(string[] args)
        {
        ② B_MyBlock MyBlock = new B_MyBlock("MyBlock_Id", "MyBlock", RS_Common.eLogLevel.NORMAL,
                                            IntPtr.Zero, RS_Common.BlockExecutionMode.DllCall);
        ③ MyBlock.MyFunction(1);
        ④ MyBlock.SendLogConsole(RS_Common.eLogLevel.NORMAL, "This is my string!");
        }
    }
}
```

```
MyApplication
  Properties
  References
    B_MyBlock
⑤   Common_cs
    CommonBaseCS
    ...
```

Figure 7-26: Direct DLL call of a block function

1 = Namespace for the B_MyBlock class
2 = Creation of a MyBlock instance with DLL call execution mode
3 = Call of the MyBlock's block function
4 = Call of a CustomerRuntimeBase function
5 = Required references (extract)

Development of a block-specific runtime package

It is even possible to override CustomerRuntimeBase functions. For this case, a block-specific runtime package has to be set up in Visual Studio. The figure shows an example of a block-specific runtime implementation for "MyBlock" where the SendLogConsole function of the CustomerRuntimeBase is overridden.

```
using System;

using RS_Common;
using Common_cs;

namespace B_MyBlockRT
{
    public class CustomerRuntimeMyBlock : CustomerRuntimeBase
    {
        // Constructor
        public CustomerRuntimeMyBlock(IBlockDefinitionBase BlockDefBase)
            : base(BlockDefBase)
        {
        }

        public override void SendLogConsole(eLogLevel LogLvl, string Text)
        {
            Console.WriteLine(Text);
        }
    }
}
```

Figure 7-27: Block-specific runtime

The namespace "<Blockname>RT" is required, here "B_MyBlockRT". The block-specific runtime class is CustomerRuntimeMyBlock which is derived from CustomerRuntimeBase. "RS_Common" and "Common_cs" have to be referenced to provide required framework functionality.

The logging methods from CustomerRuntimeBase can alternatively be overridden with the event handler technique. This technique allows to implement analogous code in an application program and a corresponding Matlab script. In the following figure, the SendLogConsole method from CustomerRuntimeBase is overridden in the B_MyBlockRT using the onRuntimeEvent method of CustomerRuntimeBase.

```
using System;

using RS_Common;
using Common_cs;

namespace B_MyBlockRT
{
    public class CustomerRuntimeMyBlock : CustomerRuntimeBase
    {
        // Constructor
        public CustomerRuntimeMyBlock(IBlockDefinitionBase BlockDefBase)
            : base(BlockDefBase)
        {
        }

        public override void SendLogConsole(eLogLevel LogLvl, string Text)
        {
            onRuntimeEvent(EventType.LOG, Text);
        }
    }
}
```

Figure 7-28: Customer runtime: Overriding a logging method with onRuntimeEvent

```
using System;

using RS_B_MyBlock;
using B_MyBlockRT;

namespace MyBlockApplication
{
    class Program
    {
        public static void LoggingCallback(object c, EventArgs e)
        {
            Console.WriteLine(((CustomerRuntimeBase.LogEventArgs)e).LogText);
        }

        static void Main(string[] args)
        {
            B_MyBlock MyBlock = new B_MyBlock("MyBlock_Id", "MyBlock", RS_Common.eLogLevel.NORMAL,
                                              IntPtr.Zero, RS_Common.BlockExecutionMode.DllCall);

            EventHandler eh =  new EventHandler(LoggingCallback);
            MyBlock.CustomerRuntime.AddLoggingEventHandler(eh);

            MyBlock.MyFunction(1);
            MyBlock.SendLogConsole(RS_Common.eLogLevel.NORMAL, "This is my string!");
        }
    }
}
```

Figure 7-29: Application program with event handler

Here, the related event handler method, LoggingCallback which actually realizes the logging, is implemented in the application program. The event handler for that method is created in the application program as well and passed to the AddLog-

gingEventHandler method (from CustomerRuntimeBase), thus addressing the event handler method to the CustomerRuntimeBase.

During test execution, logging is initiated by calling the MyBlock.SendLogConsole method. Consequently, the SendLogConsole method from the CustomerRuntime calls the onRuntimeEvent method from CustomerRuntimeBase which then executes the event handler method.

The user-defined block, the block-specific runtime and the application for calling the block function may be implemented in one Visual Studio solution. See the following figure for an example.

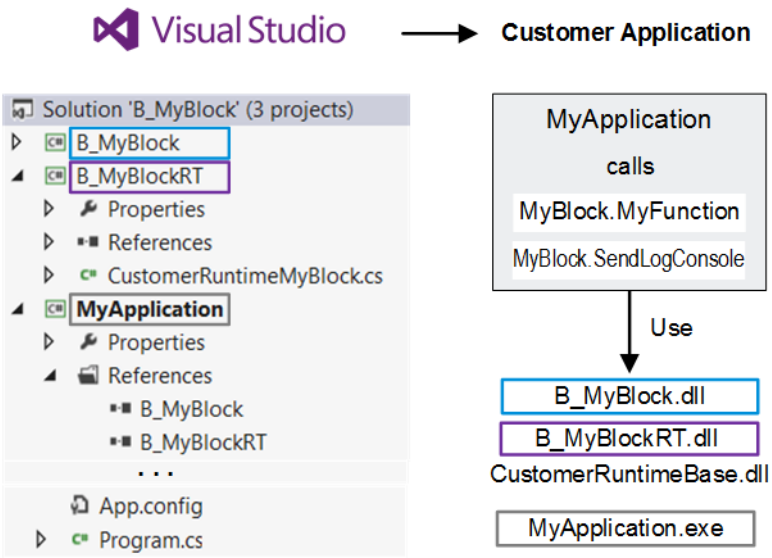


Figure 7-30: Visual Studio solution with block, block-specific runtime and application

`B_MyBlockRT.dll` must be placed next to `B_MyBlock.dll` in the file system.

`B_MyBlockRT.dll` and `B_MyBlock.dll` must be available in the DLL search path of the application.

In Visual Studio, you can adjust the output path for the `B_MyBlockRT.dll` after right-clicking the B_MyBlockRT project in the Solution Explorer and selecting "Properties". In the "Properties" view, select "Build" at the left side and enter the "Output path" in the "Output" section.

Alternatively, you can start at the MyApplication project and add a reference to B_MyBlockRT. Consequently, the output paths will be adjusted during compila-

tion, `B_MyBlockRT.dll` will be stored in the same directory as `MyApplication.exe`.

Related information (how to)

See Chapter 7.2.5, "Developing Block Functions for Direct DLL Call", on page 168.

Direct DLL call with a Matlab script

You can use your preferred programming or script language to call block functions directly from the block DLL. Here, Matlab is taken as example script language. The figure shows how "MyFunction", defined in "B_MyBlock" is called from the "B_MyBlock.dll" in a Matlab script.

```
%Required assemblies for the CustomerRuntime
NET.addAssembly('C:\Program Files\Rohde-Schwarz\QuickStep\Framework\CommonBaseCS.dll');
NET.addAssembly('C:\Program Files\Rohde-Schwarz\QuickStep\Framework\Common_cs.dll');

%Load the QuickStep block in standalone mode
NET.addAssembly('C:\Users\Public\Documents\Rohde-Schwarz\QuickStep\UserBlocks\BlockLibrary\B_MyBlock.dll');

%Instantiate the MyBlock
MyBlock=RS_B_MyBlock.B_MyBlock('MyBlock_ID', 'MyBlock', RS_Common.eLogLevel.NORMAL,
                                System.IntPtr.Zero, RS_Common.BlockExecutionMode.DllCall);

MyBlock.MyFunction(1);
```

Figure 7-31: Direct DLL block function call in Matlab

The usual logging with the provided CustomerRuntimeBase functions is not possible since the Matlab script execution already occupies the execution thread in the .NET framework. To resolve this problem, the actual logging has to be executed in Matlab. CustomerRuntimeBase logging methods, for example SendLogConsole, are overriden in the block-specific runtime file using an onRuntimeEvent method from CustomerRuntimeBase and an event handler for communication between CustomerRuntimeBase and Matlab.

```
%Create the event handler function/delegate
eh = System.EventHandler(@(o,e) disp(char(e.LogText)));

%Add the delegate to the CustomerRuntime
MyBlock.CustomerRuntime.AddLoggingEventHandler(eh);

MyBlock.SendLogConsole(RS_Common.eLogLevel.NORMAL, "This is my string!");
```

Figure 7-32: Event handler in Matlab

In the example in the figure, the Matlab script defines the logging method, here writing the log message into the Matlab console, and creates the event handler

for that method. At test execution start, Matlab passes the event handler to the AddLoggingEventHandler method, thus addressing the Matlab event handler method to the CustomerRuntimeBase. During test execution, logging is initiated by the SendLogConsole method in the Matlab script. It uses the onRuntimeEvent method from CustomerRuntimeBase. onRuntimeEvent from the overridden SendLogConsole calls the event handler method which is executed by Matlab.

7.1.8 Quality Control, Logging, Exception/Error Handling

7.1.8.1 Debugging

In the QuickStep, GUI breakpoints can be set on test steps and also on block functions by using the graphical debugger tool. If such a breakpoint is reached, block functions can be executed step-by-step.

The QuickStep runtime engine can be commanded to interrupt the test plan execution when reaching a breakpoint and to allow debugging of the code. To interrupt the test execution, a QuickStep `WaitDebug()` or `WaitDebugOnce()` function is added in the code and a usual Visual Studio breakpoint is set after the function. During test run, the QuickStep engine has to be attached to the process of the QuickStep engine once the pop-up window shows up. The test plan execution stops at each breakpoint and you can debug the code with all means provided in Visual Studio (Community). If you set several breakpoints after `WaitDebug()` in the file, the code execution will stop at each of them. Note that `WaitDebug()` is only effective for breakpoints in the same module (`.dll`, `.exe`).

It is also possible to start QuickStep directly from VisualStudio (setting the correct parameters in the startup project properties). In this case the WaitDebug() calls can be avoided and Visual Studio breakpoints are directly recognized.

How To: Chapter 7.2.6, "Debugging During Block Development", on page 172

7.1.8.2 Code, Block and System Testing

Having created new blocks or implemented new functions or built new test plans, you can do some testing of your implementation: Unit tests are provided for this purpose. Visual Studio provides the framework for unit tests and a `UnitTest.cpp` file within a project where you insert your test code. A unit test is

created once and runs every time after source code has been changed. In this way, it is assured that no bugs are introduced.

Unit testing is applicable in three stages:

- Testing functional code for a new or modified block function
- Testing a block function via the block interface, i.e. via a connection to the block
- Testing a complete test plan (system test)

QuickStep supports the unit test framework with an autocoded `UnitTest.cpp` file. For additional details please refer to the “Unit Test” chapter of the Developer Training manual.

7.1.8.3 Block Development Tool: Parameter Checks with Regular Expressions

The Block Development Tool offers syntax control with regular expressions for parameters of a string type (e.g. char[...]) in the "RegEx" field within the "Function Editor". The regular expression defines the syntax of the expected string. QuickStep checks if the entered data for the parameter in the test plan matches the regular expression. If not, the text provided in the "Error Message" field is immediately shown as warning at the GUI. The regular expression check is useful if the string parameter consists of several components, for example in case of an IP address.

Note: For the data types integer and double, minimum and maximum allowed values can be entered directly. Regular expressions do not apply to these data types.

Table 7-1: Examples for regular expressions

Example for Expected Syntax	Regular Expression	Description
GPIB::1	`^(GPIB::\b\d{1,2}\b)`	String begins with "GPIB::" followed by a one-digit number once or twice.
TCPIP::192.168.2.123	`^(TCPIP::\b(?:\d{1,3}\.){3}\d{1,3}\b)`	String begins with "TCPIP::" followed by 1-3 digit number plus dot; number plus dot comes 3 times, the string ends with a 1-3 digit number

7.1.8.4 Initial Block Check

At the beginning of each test execution, QuickStep automatically calls the Check-Block() function if "Enable Check-Block" is selected in the "TPR Options". In addition, Init() and Close() are called before and after CheckBlock() if these block functions are part of the test procedure. These calls happen before the user-defined test procedure is started. The test procedure typically calls the Init() and Close() functions again.

For each block, the CheckBlock() function is automatically generated but without any functionality. It can be used to check the firmware or hardware version of the connected instrument, for example. By default, “Enable Check-Block” is deactivated (flag is set to false).

The CheckBlock() function is contained in the file `B_[BlockName].cpp` file. The functionality has to be added manually within the CheckBlock() function via Visual Studio. The following figure shows an example of code implementation for the CheckBlock() function of the MyDcSupply block.

```
RS_ActivityBlock::ReplyCheckBlockTypePtr B_MyDcSupply::CheckBlock()
{


  // Add code to check the block functionality,
  // e.g. its associated measurement equipment (type, firmware version, option)

  // Read IDN string from instrument
  std::string IdnString = Worker->PrintIdentity();

  // check if the IDN string contains "NGMO"
  if (RS_B_CommonPublic::StringContains(IdnString, "NGMO"))
      SendLogConsole( RS_QuickStepRuntime::RS_LogLevel::NORMAL, "Check PASSED.");
  else
      SendLogConsole( RS_QuickStepRuntime::RS_LogLevel::NORMAL, "Check FAILED.");

  return CreateReplyCheckBlockTypePtr(true, "");
}
```

Figure 7-33: CheckBlock() code

The code inserted in the CheckBlock() function sends an `*IDN?` request to the test instrument via the PrintIdentity() function. It checks from the returned string if the instrument is an R&S NGMO and logs the result of the check. When executing the test plan after a new build, the log message "Check PASSED." is displayed in the "Log Viewer" if an R&S NGMO is connected.

Log Viewer

Clear Autoscroll

Timestamp	Message
2015-11-10 11:40:47.791	Start Test Plan
2015-11-10 11:40:51.369	Preparing Execution Environment...
2015-11-10 11:40:51.369	Check all Blocks...
2015-11-10 11:40:51.369	MyDcSupply Init
2015-11-10 11:40:51.494	MyDcSupply CheckBlock
2015-11-10 11:40:51.494	CheckBlock PASSED.
2015-11-10 11:40:51.494	MyDcSupply Close
2015-11-10 11:40:51.494	Initialize Execution Environment...
2015-11-10 11:40:51.494	**********
2015-11-10 11:40:51.494	********** Starting Testplan ****
2015-11-10 11:40:51.494	**********
2015-11-10 11:40:51.994	MyDcSupply Init

Figure 7-34: CheckBlock log report

7.1.8.5 Catching VISA Return Errors

The QuickStep VISA operations Write and Read return an integer status value. This return value can be evaluated to catch errors. The SetVoltage() function of the RS_PowerSupplyBase block is taken as example to show how unexpected return values can be detected and logged. The code implementation is done in the worker files.

- In `B_RS_PowerSupplyBaseWorker.h`, a `status` parameter of type ViStatus is added to the class declaration.

```
private:
    // Block definition class member
    ViStatus status;
```

Figure 7-35: Declaration of ViStatus type

- In `B_RS_PowerSupplyBaseWorker.cpp`, the actual return value of the VISA Write command (for example to set the voltage) is stored to the `status` variable and compared with the expected `status` value. If the `status` is smaller than `VI_SUCCESS`, an exception is thrown.

```
//Set channel
RS_QuickStepRuntime::Snprintf(writeStr, sizeof(writeStr), "INST OUTP%d", channel);
status |= Block->VisaStdWrite(writeStr);
//Set current
RS_QuickStepRuntime::Snprintf(writeStr, sizeof(writeStr), "CURR %2.3f", current);
status |= Block->VisaStdWrite(writeStr);
..
if (status < VI_SUCCESS)
{
    RS_B_Common::SendVisaWarning(Block, status, Block->VisaStdGetConnId(), "CURR");
}
```

Figure 7-36: Error catch for VISA return error

When running the related test plan, the desired status information can now be found in the "Log Viewer". In case of a status error, the test execution can for example be aborted using the corresponding API functions.

7.1.8.6 Structured Exception Handling

The QuickStep Runtime engine applies structured exception handling (SEH) for getting comprehensive information about exceptions during a test run. To make sure, that QuickStep can provide useful information to the user in case of errors and exceptions, the QuickStep API functions should be used to abort a test in a well defined way.

7.1.8.7 Automatic Check of Required Functions

When a test execution is started, QuickStep checks if the required functions of a user-developed block are realized as provided functions in blocks connected for inter-block communication. QuickStep considers each required function of a block separately and handles the check results in the following way:

- If a required function is not realized as provided function in a connected block, a warning is provided before the test plan is executed.
- If a required function is called during test execution but the parameters of the required function do not match the parameters of the corresponding provided function, the test execution is aborted with error report.
- If a required function is called during test execution but it is not realized as provided function in a connected block, a warning is shown.

7.1.9 Script Block Functions

QuickStep can integrate scripts of the following kinds in ScriptBlock block functions.

- Forum scripts
- Python 2.7.10 scripts (handled as Forum scripts)
- MATLAB scripts

Forum must be installed for QuickStep to support Forum or Python 2.7.10 scripts.

MATLAB must be installed for QuickStep to support MATLAB scripts.

A script block function contains one script and is used in test procedures as any other block function. Parameters for a script block function might be defined which can be used within the scripts. Handling of these parameters within QuickStep is the same as for any other block function.

Existing scripts can be re-used (after a slight adaptation to QuickStep). Alternatively, new scripts can be created or existing scripts might be modified. Forum or MATLAB is used for editing and debugging the scripts.

When a script block is called in QuickStep during a test run, the test script is executed in the same way as when executed by Forum or MATLAB.

7.1.9.1 Scripting with R&S Forum

Forum is a Rohde & Schwarz tool designed for easy and powerful remote control of test instruments with Python 2.7.10 scripts.

Core features:

- VISA-based remote scripting using Python language.
- Integrated debugger: Breakpoints, stepping through source code, inspecting variables.
- Macros: Assigning of code snippets to buttons in the GUI.
- Extensibility due to Python: Easy integration of custom Python libraries.
- Graphics: matplotlib and numpy are integrated.

New functionality, for example remote instrument control, can be implemented with Forum scripts (`*.i3e`) or directly with Python 2.7.10 scripts (`*.py`) instead

of C++ or C# programs. Forum scripts are Python scripts with a Forum-specific API extension for VISA communication. QuickStep supports this extension. Python scripts using the PyVISA library for VISA communication are also supported.

How to: Chapter 7.2.9, "Re-Using an R&S Forum Script", on page 178

Interplay of QuickStep and R&S Forum

Developing script blocks requires a certain interplay between QuickStep and Forum. The following figure shows the principle and main relations.

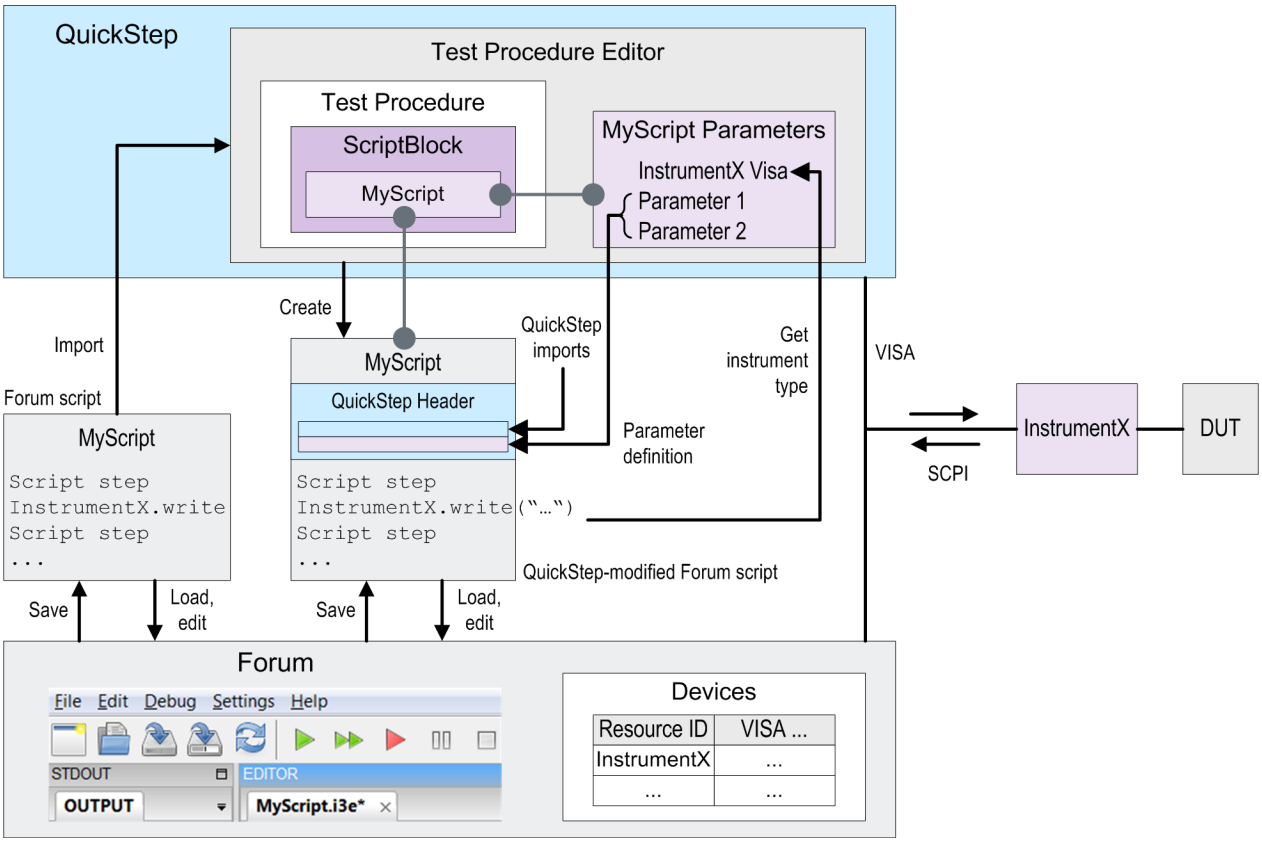


Figure 7-37: Development of script blocks with QuickStep and Forum

Main characteristics:

- Script block development can proceed from two start points:
  - An existing Forum or Python (version 2.7.10) script is imported in QuickStep, adapted to QuickStep and embedded in a script block function.
  - A new Forum/Python script template compatible with Forum is created within QuickStep and embedded in a script block function.

- A script within a QuickStep script block can be associated with parameters which are under control of QuickStep. These script parameters appear in the "Properties" area in the "Test Procedure Editor" and are handled in the same way as the function parameters of usual blocks.
- If a script is imported, QuickStep detects the VISA instruments (the used VISA alias names) to be controlled by the script. QuickStep adds the corresponding VISA resource parameters to the QuickStep script parameters. A list of the VISA aliases is also added to the QuickStep header in the script (see below). This list can be manually edited after import.
- QuickStep automatically adds a QuickStep header to a script. This header imports some Python modules for QuickStep functionality and the definition of the QuickStep script parameters (including the list of the VISA resource parameters).
  Do not change these parts. They are necessary for the availability of the QuickStep Runtime API and the needed parameters. Add code only in the indicated “Code Area”.

The resulting test script can be executed by QuickStep as part of a test procedure or directly within Forum. When the script block is called in QuickStep during a test run, the test script is executed and controls the associated test instrument in the same way as when executed by Forum. The QuickStep Python API provides QuickStep functions such as writing into the QuickStep result files or into the Log Viewer.

For further details, see the Training User manual.

7.1.9.2 Scripting with MATLAB

MATLAB is a numerical computing environment developed by MathWorks. It is a widely used tool especially for calculations with arrays and matrices. Complex data is natively supported as well. A big variety of toolboxes is available to extend the functional scope of MATLAB.

Interplay of QuickStep and MATLAB

Main characteristics:

- Existing scripts can be imported or new ones can be created.
- A script within a QuickStep script block can be associated with parameters which are under control of QuickStep. These script parameters appear in the "Properties" area in the "Test Procedure Editor" and are handled in the same way as the function parameters of usual blocks.

- QuickStep automatically adds a QuickStep header to a script.
  Do not change these parts. They are necessary for the availability of the QuickStep Runtime API and the needed parameters. Add code only in the indicated “Code Area”.
- Session selection by QuickStep when the script block is called during a test run:
  – In case no MATLAB session is running, QuickStep automatically opens the MATLAB desktop user interface (which stays open after completed test execution) and the requested MATLAB script.
  – In case a MATLAB session is already running with COM enabled, QuickStep attaches to the first instance with this setting. Note that COM is not enabled by default. COM is enabled with the following command:
  `enableservice('AutomationServer',true);`
  – In case a MATLAB session is already running with COM not enabled, QuickStep starts its own MATLAB session and initializes the settings accordingly.

Opening of the MATLAB GUI and its initialization take a few seconds. If you do not want to lose time during test procedure execution, create an empty MATLAB script with QuickStep and insert it in the "Test Procedure Before" execution phase. So, a MATLAB session is started beforehand.

For further details, see the Training User manual.

7.1.10 User-Defined GUI

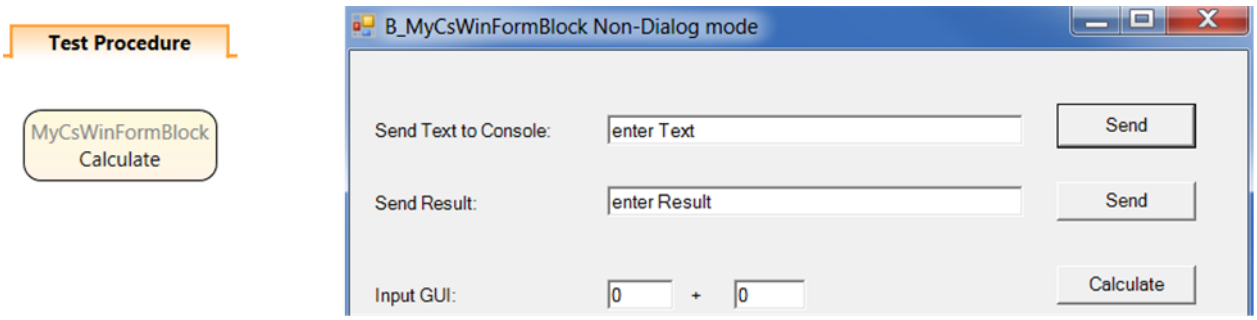


Figure 7-38: Windows Forms block function and related dialog (example)

QuickStep can integrate a user-defined dialog in a test. Such a dialog, also called application GUI, is a window with interactive elements (text boxes, buttons, ...)

started during test execution. To provide a dialog, a user-defined dialog block has to be created and dialog block functions have to be added in the test procedure. The dialog elements are implemented in Visual Studio with Windows Forms or WPF under C#.

A dialog allows the user to control a test to a certain degree during test execution instead of adapting the test procedure beforehand. Operators can use one predefined test procedure for different test cases and avoid editing of the test procedure if the dialog provides the required adaptation options. Dialogs are particularly helpful if you combine them with test plan execution by shortcut (created via "File > Create Testplan Execution Shortcut" from the QuickStep menu): The operator does not need to open and use the QuickStep GUI. He just clicks the shortcut and controls the running test with the dialog.

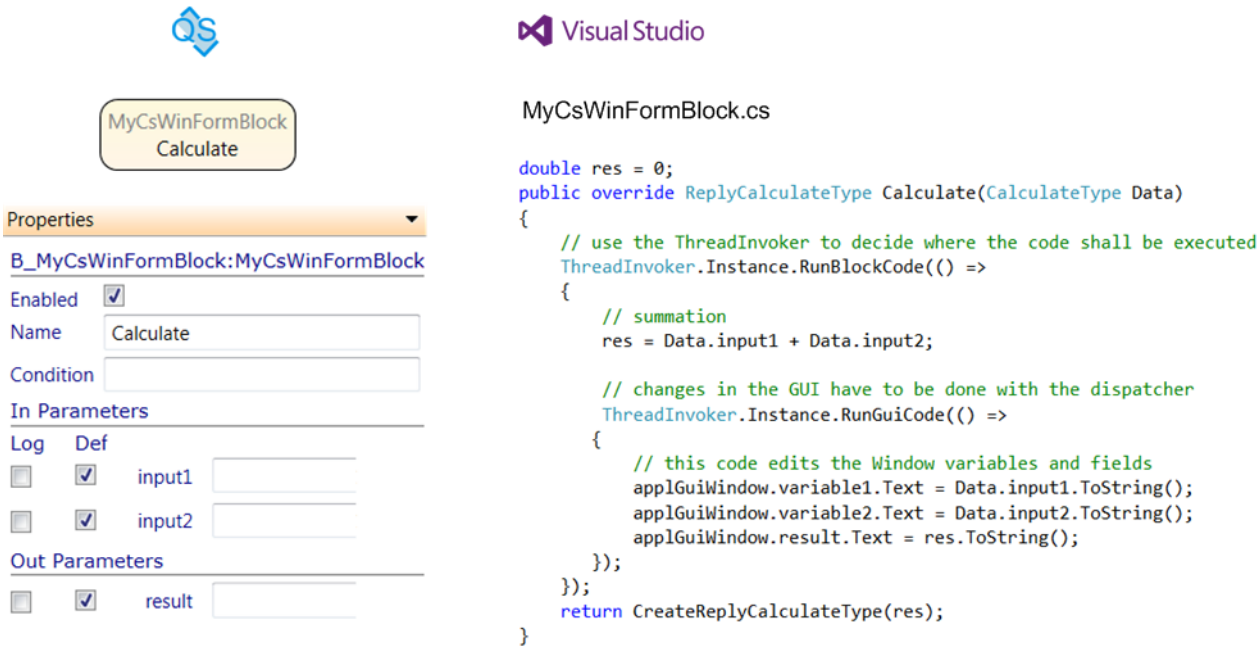


Figure 7-39: Properties of a dialog block function and related implementation (example)

Dialog blocks are developed similarly as other user-defined blocks. First, the Block Development Tool is used to create a new block. In this step, a block template has to be selected which specifies the type of dialog (see below for details). Second, you define block functions and parameters as usual leading to a block definition. Then you export the block definition and get a block project. QuickStep automatically implements a default dialog in this step. Eventually, you open the block project in Visual Studio, adapt and implement the dialog details and implement block functions defined in the Block Development Tool.

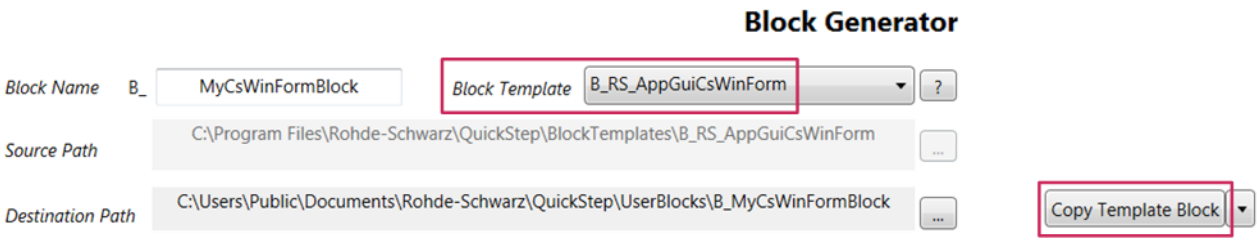


Figure 7-40: Dialog block creation with template in the Block Development Tool

Dialog block types defined by the block templates:

- "B_RS_NonBlockingGuiCsWinForm": Windows Forms dialog block based on C# and with own dialog thread
- "B_RS_NonBlockingGuiCsWpf": WPF dialog block based on C# and with own dialog thread
- "B_RS_BlockingGuiCsWinForm": Windows Forms dialog block based on C# and with one shared block thread
- "B_RS_BlockingGuiCsWpf": WPF dialog block based on C# and with one shared block thread

In case of a a shared block thread, only one block function is executed at a particular time. If a dialog is running in this mode, an additional execution request for another function of this block has to wait until the dialog has been closed ("Blocking GUI").

A dialog with own thread does not block other block activities while the dialog is running. For example, the block can receive information which is used for updating the dialog.

Programming structures

The code for a dialog block is distributed over several files. Main files and selected dialog-related content:

- `ApplGuiWindow.Designer.cs`: Contains the dialog element definitions.
- `ApplGuiWindow.cs`: Contains the event (button clicks etc.) implementations. These are related to thread invocations in case of a dialog with own thread.
- `B_<Block Name>.cs`: Contains the implementation of the functions to be executed via dialog commands. In case of a dialog with own thread also contains functions for creating, starting and closing the dialog thread.

See the figure for a selection of example code.

ApplGuiWindow.Designer.cs

```
this.textBox3 = new System.Windows.Forms.TextBox();
this.textBox4 = new System.Windows.Forms.TextBox();
this.label4 = new System.Windows.Forms.Label();
this.button4 = new System.Windows.Forms.Button();
          ...
this.button4.Location = new System.Drawing.Point(381, 125);
this.button4.Name = "button4";
this.button4.Size = new System.Drawing.Size(75, 23);
this.button4.TabIndex = 11;
this.button4.Text = "Calculate";
this.button4.UseVisualStyleBackColor = true;
this.button4.Click += new System.EventHandler(this.button4_Click);
```

ApplGuiWindow.cs

```
private void button4_Click(object sender, EventArgs e)
{   // call the BlockFunction from the GUI Thread
    double in1, in2;
    double.TryParse(this.textBox3.Text.ToString(), out in1);
    double.TryParse(this.textBox4.Text.ToString(), out in2);
    ThreadInvoker.Instance.RunBlockCode(() => BlockInstance.Calculate(in1,in2));
}
```

B_MyCsWinFormBlock.cs

```
public override ReplyOpenType Open(OpenType Data)
{   // create a new window thread to show it parallel
    STAThread = new Thread(() => StartGuiThread(Data));
    STAThread.SetApartmentState(ApartmentState.STA);
    // the started thread must be joined later
    STAThread.Start();
    // wait for the creation of the window -> should be
    Thread.Sleep(500);
    return CreateReplyOpenType();
}
```

Figure 7-41: Code examples

Test Procedure for a dialog with own thread

The Open function of the dialog block creates the dialog thread. So, this block function is mandatory in the "Test Procedure Before" phase. Use the Close block function as well.

7.2 Procedures Related to Block Development

This chapter provides step-by-step descriptions of how to handle the main block development tasks. The descriptions focus on confined problems and compact solutions.

Another source for procedure descriptions is the **Training manual**. It uses instructive examples as starting points and provides extensive information including aspects of motivation, concept, code examples and illustration of the results. Examples of Training manual content:

- Extending the functionality of an existing block
- Using regular expressions

- Details on using the System Configurator, for example creating graphical symbols

7.2.1 Overview: Implementation of New Functionality

The development of a user-defined block and its integration in a test project starts with the Block Development Tool, proceeds to Visual Studio and is completed in QuickStep. Subsequent modifications usually take the same route resulting in a development cycle.

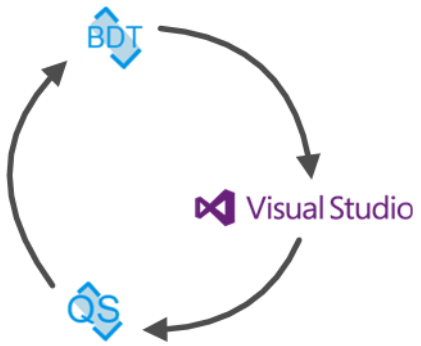


Figure 7-42: Development cycle

The provision of new functionality comprises the following basic steps:

Using the Block Development Tool

- Creating a new block and defining its basic properties.
  - Providing the name for the generated Visual Studio (Community) block project
  - Defining the block functions (provided and required) and their parameters
- If necessary, adding functions and parameters to existing user-defined blocks
- Exporting the block definition to code

Using Visual Studio (Community)

- Building the block project generated by the Block Definition Tool. This creates a solution for the block and generates the definition files `B_[BlockName]Definition.h` and `B_[BlockName]Definition.cpp`
- Implementing the user-defined functions
- Building the completed solution again (resulting in a `*.dll` file for QuickStep)

Using QuickStep

- Updating the current test project (or closing and reopening the GUI). QuickStep detects the new `*.dll`. Now the new block is available at the GUI
- Creating a test procedure with the new block
  - Dragging the new block into the "Blocks & Connectivity" tab
  - Setting up the Init and the PrintIdentity block functions in the "Testrun Before" phase
  - Selecting the user-defined functions in the "Test Procedure" phase
  - Setting up the Close block function in the "Testrun After" phase
- Configuring the test project
  - Updating the test project to apply the changes in the test procedure
  - Connecting the test steps with the test procedure

If functions for a user-defined block have to be added or modified afterwards, another walkthrough from Block Development Tool to Visual Studio and to QuickStep is required.

The following procedures describe details and also provide information about optional tasks.

Code preservation if a block definition is modified

Suppose that a user-defined block whose functions have already been implemented with Visual Studio is loaded in the Block Development Tool (BDT). If the block properties are modified, saved and exported, the already implemented source code is preserved. Examples:

- Adding a parameter to a block function of an already implemented block in the BDT: The `*.cpp` files are not affected at all. Instead the new parameter appears only in `B_...Definition.h` where the parameter structures are defined.
- Adding a block function to an already implemented block in the BDT: The skeleton for the new block function is added in the existing code files. The source code of the already existing functions is not affected.
- Removing a block function from an already implemented block in the BDT: When the "Save & Export Project" is pressed, a notification dialog pops up stating that the block definition and the source code are not compatible. "Save & Export Project" is executed without deleting the incompatible source code.

The code of the removed block function has to be deleted manually. This avoids that user code is deleted unexpectedly by the BDT.

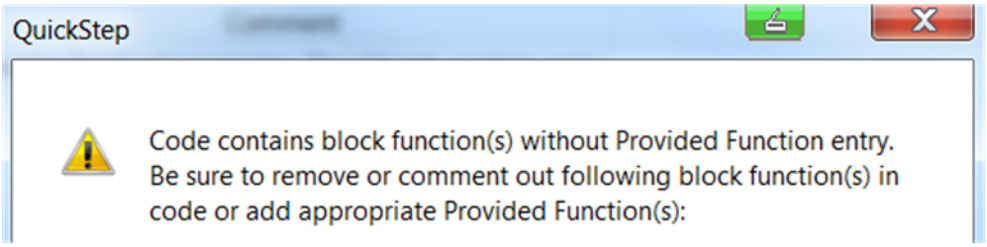


Figure 7-43: Incompatible modification of a block definition

7.2.2 Creating a New Block and Adding a Function

Only the R&S Block Development Tool is required. Proceed as follows:

1. Starting from the Windows Start icon and "All Programs", open the "QuickStep" folder, then the "Tools" folder and click "R&S QuickStep Block Development Tool".

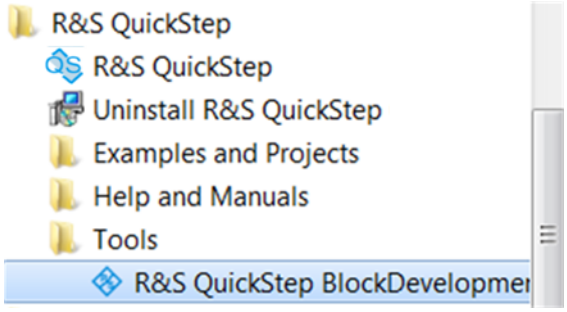


Figure 7-44: Block Development Tool: Start

The tool starts up and displays the "R&S Block Development Tool" window.

2. In the "Block Generator" section:
   - Enter a name for the block.
   - Select the block type ("Instrument Block with VISA" if the block shall control a test instrument over a VISA interface).
   - If you like, modify the destination path where the new block is stored. You can also navigate to the desired directory with the "..." button.
   - Select the programming language ("C++" or "C#") and the Visual Studio version used for programming.

Figure 7-45: Block Development Tool: Create new block

3. Click the "Create New Block" button.
   A new block with some default functions and default properties is created. The block is automatically loaded and appears on the left side of the "Block Definition File Editor" area.

4. With the "Provided Functions" tab in the foreground, click the "new Function" button to add a "newFunction" in this area.

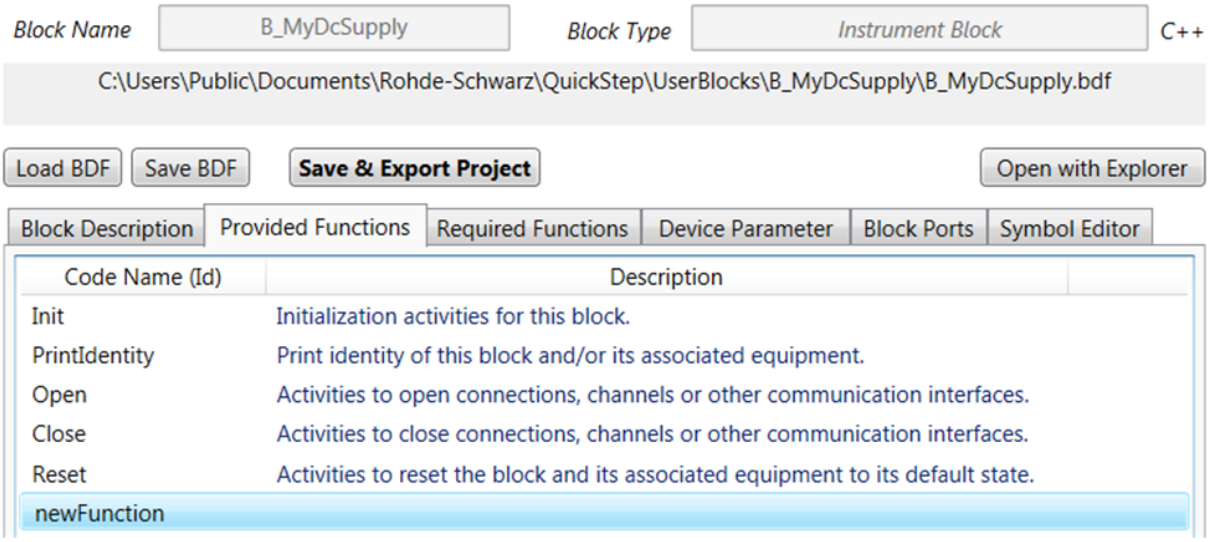


Figure 7-46: Block Development Tool: New block function

5. Double-click the "newFunction" to open the related "Function Editor".

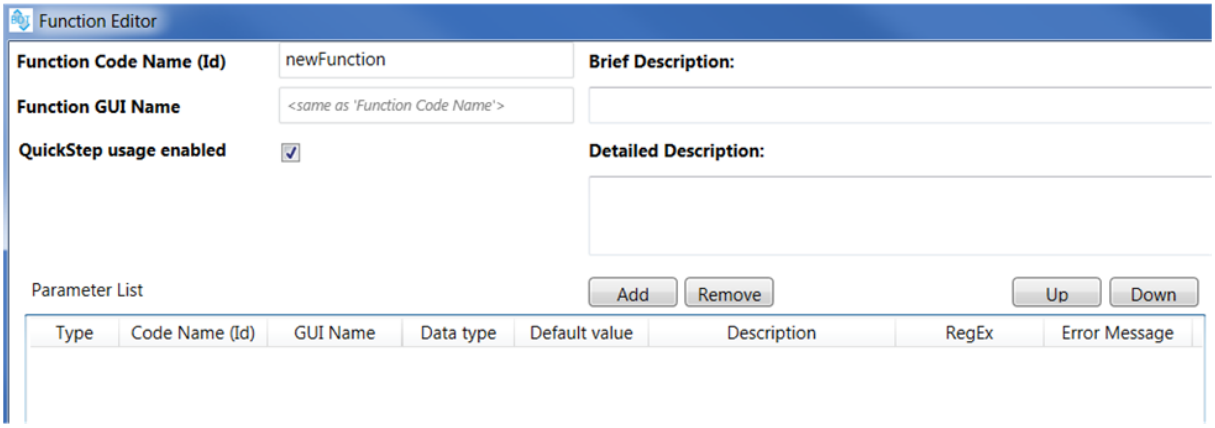


Figure 7-47: Block Development Tool: Function editor

6. Rename the function in the rows on top of the editor.
7. Click the "Add" button on top of the "Parameter List" section to add a parameter for the function in the list.
8. Set the parameter properties.

   Mandatory parameters:
   - "Code Name (Id)": The block function name used in the C++ code.
   - "GUI Name": The name which is displayed in the GUI. Choose a self-explanatory name if possible. By default, the GUI Name is identical to the Code Name.
   - "Data type".

   See Table 9-22 for a description of all parameters.
9. To provide an enumeration list of values for a parameter:
   - Click the row which describes the parameter in the "Parameter List" table.
   - Then click "Add" above the Enum List area on the right. Click "Add" as many times as enumeration values are needed.
     For each "Add", a pair of "Item Name (GUI)" and "Value" input fields is created.
   - Enter the "Item Name (GUI)"s and "Value"s in the Enum List.
     The item name appears as selectable list item for the parameter at the GUI. The "Value" entry is used for test execution. This value must comply with the selected data type for the parameter.

   Note: A parameter is automatically defined as enumeration type after at least one Enum List entry has been created. In the "Testplan Editor", a parameter's enumeration list is displayed as a drop-down box for selecting a parameter value.

10. Click "OK" to close the "Function Editor".
11. Click the "Save & Export Project" button.

Result: A block folder is generated with the entered name and under the selected path. This folder contains the `B_[BlockName].bdf` block definition file and programming structures. Project (`*.vcxproj`) and compilation files (e.g. files describing the dll interface) are also included in the block. The programming structures consist of `*.h` header files and associated `*.cpp` source code files.

Regarding further processing with Visual Studio, a block folder forms a project.

Please refer to the Developer Training manual on details how to implement specific functionality in the code for a block using the provided QuickStep API.

7.2.3 Calling a Block Function from Another Block

This procedure shows how the calling of a function from a certain block is implemented in the calling block and how the calling is realized in a test procedure in QuickStep. The following example is taken:

- There is a MyGenerator block including the GetGenPowerLevel function which returns the Out parameter GenPowerLevel. The GetGenPowerLevel function has an In parameter of string type.
- There is a MyAnalyzer block including the SetReferenceLevel function which shall call the GetGenPowerLevel function and shall output the return value in the log file.

It is assumed that the MyGenerator block and its functions have already been defined in the Block Development Tool and the related functional code has been implemented in Visual Studio. The MyAnalyzer block has also been created but without considering the GetGenPowerLevel() function to be called. For convenience reasons, the SetReferenceLevel function is used in an existing test project and test procedure. The test plan is already connected with the test procedure.

1. In the Block Development Tool, load the B_MyAnalyzer block.
2. On the right side of the "Block Definition File Editor" area, click the "Load" button.
   A Windows Explorer is opened.
3. Navigate to the `B_MyGenerator` folder.

4. Select the `B_MyGenerator.bdf` block definition file and open it.
5. Activate the GetGenPowerLevel function on the right side via check box.
6. Make sure that on the left side the "Required Functions" tab is selected. Then click the "<---" button in the middle of the "Block Definition File Editor".
   The selected function is copied to the "Required Functions" on the left.

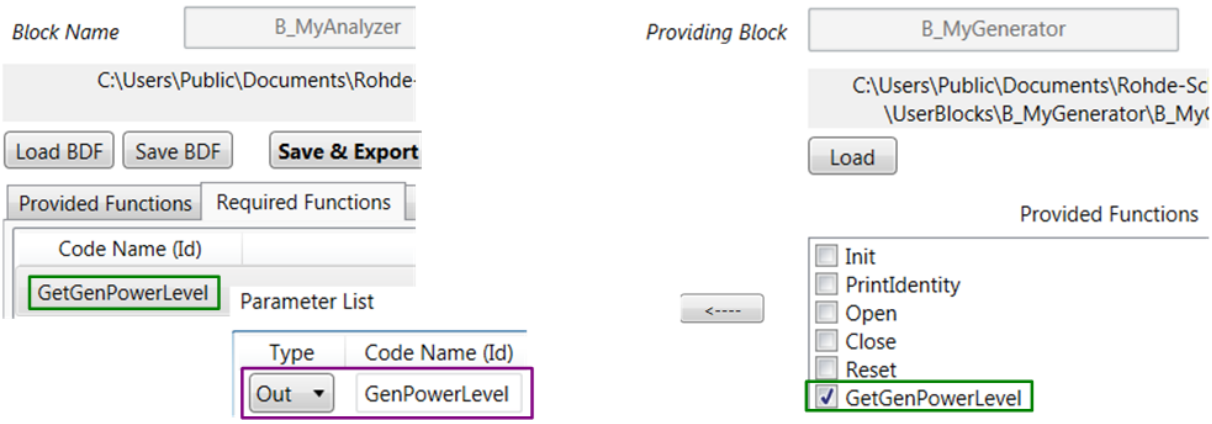


Figure 7-48: Block Development Tool: Declaring the required function

7. If you like, activate the "Block Ports" tab and change the Id of the B_MyAnalyzer's out port which will be used in the code for calling the GetGenPowerLevel() function. Here, the default Id ("BlockOutPort") is kept.
8. Click "Save & Export Project".
9. Go to Visual Studio, open the B_MyAnalyzer project and have the `B_MyAnalyzer.cpp` file displayed.
10. Navigate to the SetReferenceLevel() function which shall call the GetGenPowerLevel() function and enter the calling code.

```
ReplySetReferenceLevelTypePtr B_MyAnalyzer::SetReferenceLevel(SetReferenceLevelTypePtr &Data)
{

  // Add your code here

  ...
  ReplyGetGenPowerLevelTypePtr Reply =  SendSyncGetGenPowerLevel(B_MyAnalyzerPorts.BlockOutPort);

  SendLogConsole(RS_QuickStepRuntime::RS_LogLevel::NORMAL, "Received GenPowerLevel: %f", Reply->GenPowerLevel);

  // Be sure to fill the reply type ptr appropriately
  return CreateReplySetReferenceLevelTypePtr();
}
```

Figure 7-49: Synchronous call with reply in ...MyAnalyzer.cpp

The `SendLogConsole(...)` command is added to get a report in the Log Viewer and to see which value was received from the called function.

11. Build the B_MyAnalyzer project. Make sure that the build finished without errors.
12. Go to QuickStep and open the desired test project.
13. In the "Test Procedure Editor", select the "Blocks & Connectivity" phase.
14. Drag and drop one MyAnalyzer block and one MyGenerator block into the main area.
    Consequently, these blocks will be available for the different test execution phases (like "Testrun Before").
15. Draw a line from the MyAnalyzer's out port to the MyGenerator's in port.
    The resulting connection is used for the inter-block communication.

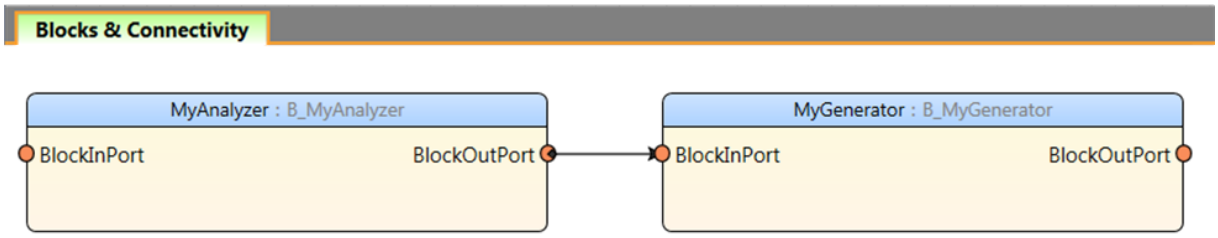


Figure 7-50: QuickStep: Drawing the connection for inter-block communication

16. Select the "TestProcedure" phase.
17. Drag the block function MyAnalyzer > SetReferenceLevel from the "Library" into the main area. Integrate the block into the existing test procedure according to your needs.
    Note that the GetGenPowerLevel does not appear in the test procedure. It will be called when SetReferenceLevel is executed. This call is part of the code implementation in SetReferenceLevel but not of the test procedure.
18. Go to the "Test Plan Editor" tab and click the "Update Test Project" button on the menu bar.
19. Adjust the test plan as you like.
20. Start the test execution.

Result: The GetGenPowerLevel function is called when SetReferenceLevel is executed. The execution of the SetReferenceLevel and the effects of calling GetGenPowerLevel are reported in the "Log Viewer" and in the Execution Protocol in the "Results Viewer".

Source	Rep/TestStep		Block	Log
Source	Rep	TestStep	Block	Log
Source	Rep	TestStep	Block	Log
LogViewer	0	0	Sequencer	Initialize Execution Environment...
LogViewer	0	0	Sequencer	**********
LogViewer	1	1	QuickStepEngine	Running Testprocedure: 'DoReferenceLevel'
LogViewer	1	1	QuickStepEngine	Starting TestStep 1. Starting Time: 2016/04/04;
LogViewer	1	1	MyAnalyzer	B_MyAnalyzer SetReferenceLevel
LogViewer	1	1	MyGenerator	B_MyGenerator GetGenPowerLevel
LogViewer	1	1	MyAnalyzer	Received GenPowerLevel: 10.000000
LogViewer	1	1	QuickStepEngine	TestSteps completed!

Figure 7-51: Results Viewer: Log of function call

7.2.4 Using Device Parameters

This procedure shows how to define a device parameter and to modify the implemented code for avoiding unnecessary transmissions of SCPI commands. Any other time consuming activity can also be avoided which shall not be executed if parameter values do not change. The user-defined block `B_MyDcSupply` is taken as example. It contains the `SetCurrent(Current)` function which sets the value of the `Current` parameter for the associated device with a SCPI command.

Starting situation: QuickStep is closed and the R&S Block Development Tool is opened. It is assumed that the `MyDcSupply` project has been built before.

The `SetCurrent(Current)` function shall be implemented in the `MyDcSupplyWorker.cpp` worker file (this is no restriction).

1. At the R&S Block Development Tool, load the `B_MyDcSupply.bdf` block definition file from the `...\UserBlocks` project directory.
2. Select the "Device Parameter" tab.
3. Create a new parameter by clicking the "new Device Parameter" button.
4. Double-click the "newParameter" to open the "Device Parameter Editor".
5. Enter the properties for the parameter (name, data type, ...).

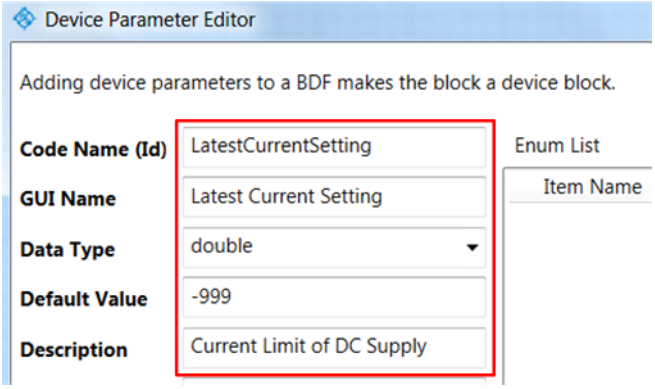


Figure 7-52: Definition of the device parameter LatestCurrentSetting

6. Click "OK" and then "Save" to update the block definition file.
7. Open the `MyDcSupply` project in Visual Studio. Build the project again to have the automatic coding steps executed. Make sure that the build succeeds without errors.
8. Open `B_MyDcSupply.cpp`.
9. Navigate to the `Init()` function and insert the `SetForceCommand(...)` method with parameter value `false`.
   This method sets the ForceSCPI flag to false which enables the mechanism for skipping SCPI command transmissions.

```
ReplyInitTypePtr B_MyDcSupply::Init(std::string PortId, InitTypePtr &Data)
{
 …
  // **** USER CODE START ****

  // set ForceSCPI flag to false
  SetForceCommand(false);

 …
  return CreateReplyInitTypePtr((char*)result_Init.c_str());
}
```

Figure 7-53: ForceSCPI flag

10. Go to the `SetCurrent()` function in the `B_MyDcSupplyWorker.cpp` file.
11. Enter device parameter functions as shown in the figure and save your modifications.

```
void B_MyDcSupplyWorker::SetCurrent(double Current)
{
    char WriteStr[DEFAULTBUFSIZE];

    // Check if new value has to be programmed
    if (Block->IsDirtyOrDiffOrForcedLatestCurrentSetting(Current))
    {
        // send SCPI command
        RS_QuickStepRuntime::Snprintf(WriteStr, DEFAULTBUFSIZE, ":SOUR:CURR %lf", Current);
        Block->VisaStdWrite(WriteStr);
        // set Device Parameter to new value
        Block->SetValueLatestCurrentSetting(Current);

        // for visualization only:
        // send a log into the LogViewer to see if SCPI was sent or not
        Block->SendLogConsole( RS_QuickStepRuntime::RS_LogLevel::NORMAL, "SetCurrent SCPI sent.");
    }
}
```

Figure 7-54: Implementation of the device parameter mechanism

12. Build the project again and make sure that the build succeeds without errors.

Result: The device parameter functionality is integrated in the block and gets active when the block is called during test run.

For testing if unnecessary SCPI command transmissions are avoided, setup a test plan with a number of test steps and a test procedure where `SetCurrent()` is used. Take care that the value for the "Current" parameter under "SetCurrent" is only changed in a few steps. Then run the test plan and check the number of SCPI commands for "SetCurrent" appearing in the "Log Viewer" or in the `ExecutionProtocol.txt` file.

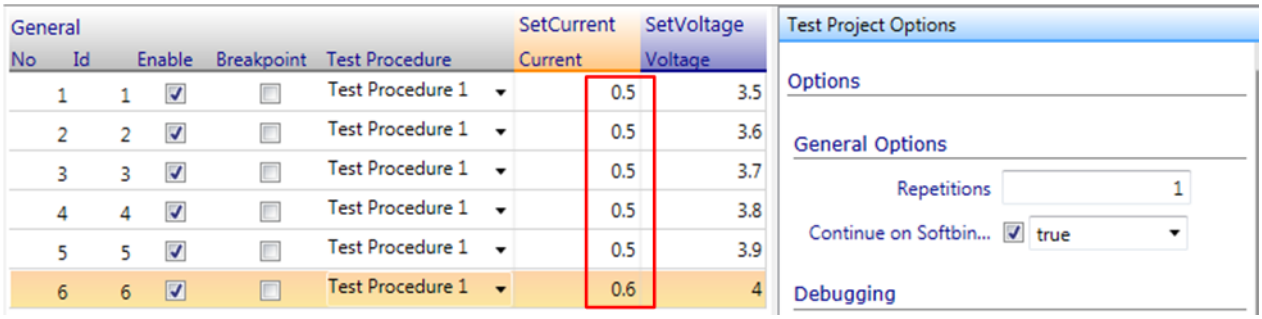


Figure 7-55: SetCurrent in test plan

7.2.5 Developing Block Functions for Direct DLL Call

Direct DLL call means that block functions are called from the block DLL without using QuickStep. QuickStep is only used for block development but not during block execution time.

The block development and provision of the application on the target system proceeds in several steps:

- To create block functions for direct DLL call
- To implement block functions for direct DLL call
- "To override CustomerRuntimeBase functions" on page 169
- To create a C# program for controlling the block functions
- "To provide the DLL solution on the target system" on page 172

<B_MyBlock> is taken as example and stands for any customer-defined block name.

See Chapter 7.1.7, "Block Functions for Direct DLL Call", on page 137 for concept information.

To create block functions for direct DLL call

Block functions for direct DLL call are created in the Block Development tool. C# is required as programming language.

1. Open the Block Development Tool ("Start > All Programs > R&S QuickStep > Tools > R&S QuickStep BlockDevelopmentTool").
2. In the "Block Generator" section, create a new block with Programming Language C#.
   The new block is automatically loaded in the "Block Definition File Editor" section.
3. In the "Provided Functions" tab of the "Block Definition File Editor" section, create a new function and double-click the function.
4. In the "Function Editor", activate the check box at "Direct Dll call enabled".

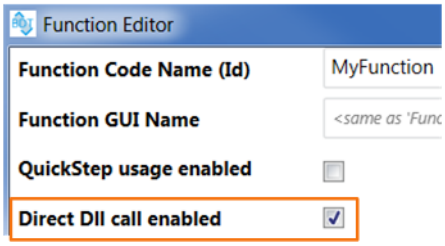


Figure 7-56: Direct DLL call enabled

5. Configure the function parameters according to your needs and click "OK" to save the settings.
6. Create and configure further functions for direct DLL call in the same way.
7. Back in the "Block Definition File Editor", click "Save & Export Project".

   A block folder is generated with the entered name and under the selected path. This folder contains the `<B_BlockName>.bdf` block definition file and programming structures. Project (`*.vcxproj`) and compilation files are included in the block. The functions for direct DLL call are exported in the `<B_MyBlock>DirectCallInterface`.

To implement block functions for direct DLL call

Code implementation is done in the same way as for block functions to be used with QuickStep during test execution. You may use functions of the Customer-RuntimeBase DLL which will be included in the solution as embedded resource.

1. Open Visual Studio and load the project ("File > Open Project", then select the `*.vcxproj` project file for the block).
2. In the Solution Explorer, select the `<B_MyBlock>.cs` programming file to load it in the main pane.
3. If you would like to use CustomerRuntimeBase functions, add `using common_cs;` in the header section.
4. Implement the functionality for the new functions in the usual way.

To override CustomerRuntimeBase functions

The CustomerRuntimeBase functions are overridden in a `<B_MyBlock>RT` project which has to be created manually. Naming convention: The name of this project is the name of the block project with `RT` (for runtime) appended. The block-specific runtime project will be referenced in the application program.

Starting point: The block project has already been loaded in Visual Studio.

1. In the Solution Explorer, right-click the solution "<B_MyBlock>", select "Add" and then "New Project".
2. In the "Add New Project" dialog, select "Visual C#" and "Class Library". In the bottom section, enter "<B_MyBlock>RT" (block project name with RT appended) for the "Name" of the new project and then click "OK".
   The project is created and its programming file is opened.
3. Add `using RS_Common;` and `using Common_cs;` in the header section.

```
using System;

using RS_Common;
using Common_cs;

namespace B_MyBlockRT
{
    public class CustomerRuntimeMyBlock : CustomerRuntimeBase
    {
        // Constructor
        public CustomerRuntimeMyBlock(IBlockDefinitionBase BlockDefBase)
            : base(BlockDefBase)
        {
        }

        public override void SendLogConsole(eLogLevel LogLvl, string Text)
        {
            Console.WriteLine(Text);
        }
    }
}
```

Figure 7-57: <B_MyBlock>RT program file for overriding CustomerRuntimeBase functions

4. Derive your runtime class with any name from CustomerRuntimeBase.
5. Enter the constructor for your runtime class derived from base.
6. Override CustomerRuntimeBase functions according to your needs.

   Tip: Select `CustomerRuntimeBase` and press the F12 key to have the CustomerRuntimeBase functions displayed which can be overridden.
7. Save all files.
8. Build the solution.
   If errors occur, correct the code as indicated.

To create a C# program for controlling the block functions

Here, for convenience, the application program for controlling the block functions is created in the block solution. Separate development of the application program in a different programming language or in a script language is also possible.

1. In the Solution Explorer, right-click the solution "<B_MyBlock>", select "Add" and then "New Project".
2. In the "Add New Project" dialog, select "Visual C#" and the application type, for example "Console Application". In the bottom section, enter an appropriate "Name" and then click "OK".
3. The application project is created and its `Program.cs` file is opened.
4. Include `using RS_<B_MyBlock>` in the header section.
5. If a block-specific runtime project with namespace `<B_MyBlock>RT` has been created, add `using <B_MyBlock>RT;` in the header section.
6. Create an instance of the block including that the `BlockExecutionMode` is set to `DllCall`.

```
using System;

using RS_B_MyBlock;
using B_MyBlockRT;

namespace MyBlockApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            B_MyBlock MyBlock = new B_MyBlock("MyBlock_Id", "MyBlock", RS_Common.eLogLevel.NORMAL,
                                              IntPtr.Zero, RS_Common.BlockExecutionMode.DllCall);
            MyBlock.MyFunction(1);
            MyBlock.SendLogConsole(RS_Common.eLogLevel.NORMAL, "This is my string!");
        }
    }
}
```

Figure 7-58: Application program

7. Call block functions in the program according to your needs.
8. In the Solution Explorer, add the following references to the application project (right-click the "Reference" folder in the application project and select "Add Reference"):
   - B_MyBlock
   - B_MyBlockRT (if a block-specific runtime is used)

- Common_cs
- CommonBaseCS

Alternative: To create a Matlab script for controlling the block functions

The structure of the Matlab script is the same as for the C# program. Additionally, event handling is required for logging.

► See "Direct DLL call with a Matlab script" on page 144 for details.

To provide the DLL solution on the target system

1. Copy the `<B_MyBlock>.dll` and `<B_MyBlock>RT.dll` (if used) into the program folder of the target system.

   Note: Make sure that both DLLs are located in the same folder.
   Make sure that the DLLs are located in the DLL search path of the application.

2. Add your application for controlling the block functions on the target system.

7.2.6 Debugging During Block Development

This procedure shows how to debug a user-developed block with the Visual Studio Debugger in an easy and safe way. The Debugger is started in Visual Studio directly from the block project.

The QuickStep "NewBlock" block is used as example leading to the Visual Studio project "B_NewBlock". Coding is done in C++ (analogous steps for C#). Starting situation is that code for the block has been modified in Visual Studio and shall be debugged now.

1. In Visual Studio, build the block project and verify that the corresponding `*.dll` is created.

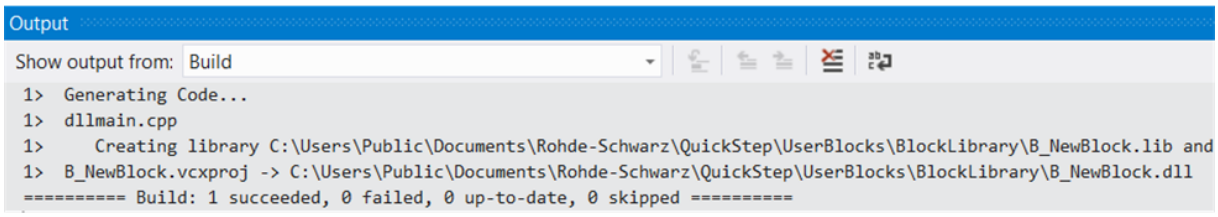


Figure 7-59: Visual Studio, build report

The block is available in QuickStep.

2. In QuickStep, open an appropriate test project (or create a new one), create a test procedure with the new block and then a test plan which uses the block.
3. Select "File > Save Test Project".
   The test project is saved. The `*.tpl` test plan file is included.
4. Temporarily save the path to the test plan including the file name (for later usage in step 8).
5. Back in Visual Studio with the block project to be debugged still opened: Right-click the block project in the "Solution Explorer" and select "Set as StartUp Project" (or select "PROJECT > Set as StartUp Project" from the menu).

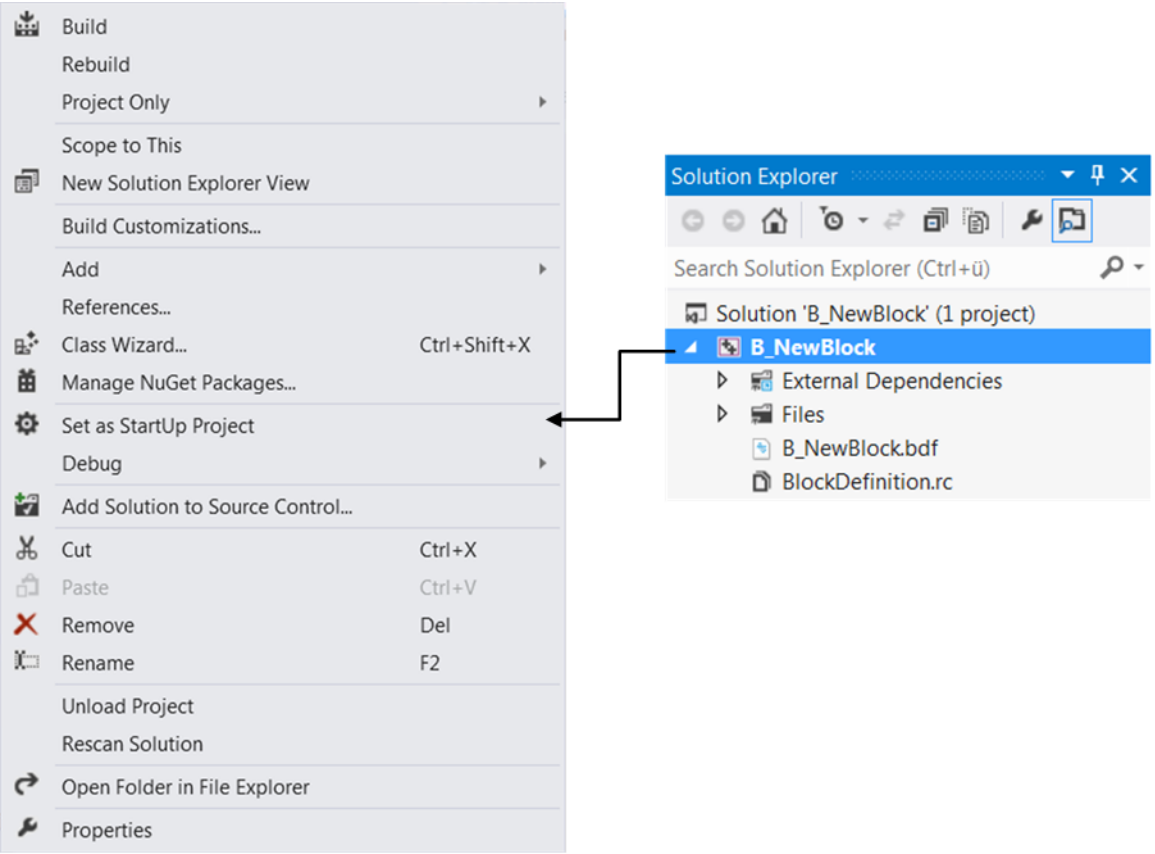


Figure 7-60: Visual Studio, project shortcut menu

6. Right-click the block project again and select "Properties".
7. In the "Properties" dialog, select "Debugging" in the navigation area on the left side.
8. Enter the following information in the main area:

- "Command": Path to the `QuickStepEngine.exe` file (including file name, see the figure)
- "Command Arguments": Path to the saved test plan (including test plan file name and file extension `.tpl`)
- "Debugger Type": You can keep the "Auto" default setting

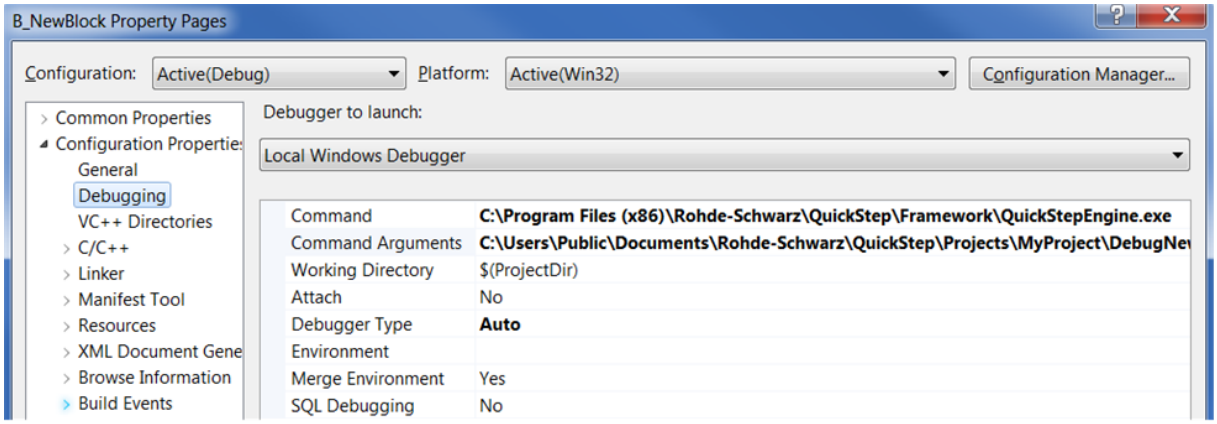


Figure 7-61: Visual Studio, properties of a block project

9. Click "OK" to save the settings and close the dialog.
10. Open the file within the block project to be debugged and set breakpoints in the code via the shortcut menu.
    Setting a breakpoint in a constructor is the safest way to start debugging.
11. Build the project.
12. Press [F5] to start debugging at the very beginning of the code execution.
    The debugger halts at the first breakpoint (if not earlier in case of an error).
13. Check if the debug symbols were loaded correctly.

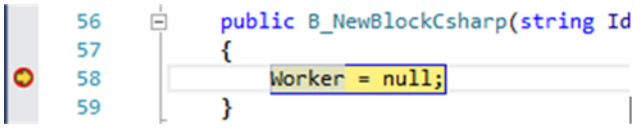


Figure 7-62: Visual Studio, debug symbol

If the symbols are not available, change the working directory or set the debug symbols path explicitly.

7.2.7 Debugging During Test Execution

This procedure shows how to set breakpoints in the source code and how they are put in operation in QuickStep.

Starting situation: `B_[Blockname].cpp` has been opened in Visual Studio (Community) at the code section where you want to start debugging.

Proceed as follows:

1. Add the `WaitDebug()` execution breakpoint in the code as shown in the following figure.

```
285
286  ReplyCalculateTypePtr B_Calculator::Calculate(CalculateType* &Data)
287  {
288      // Add your code here
289      double result;
290
291      //set execution breakpoint (stops only once)
292      RS_QuickStepRuntime::WaitDebugOnce();
293
294      // Create an output into the LogViewer of the GUI which states the selected operation
295      std::string str(Data->OPMODE);
296      SendLogConsole(RS_QuickStepRuntime::RS_LogLevel::NORMAL,"Operation: %s",RS_B_CommonPub
297
```

Figure 7-63: Setting an execution breakpoint

2. Add a Visual Studio breakpoint as desired for a subsequent code line: Right-click the desired line and select "Breakpoint > Add Breakpoint" in the context dialog.
   A red circle on the left of the code line indicates the breakpoint (line 295 in the figure).
3. Compile the project.
4. Start QuickStep (which takes into account the compiled `*.dll` of the previous step). Then start the test plan containing calls of the block functions with the breakpoints from the QuickStep GUI.
5. When the `WaitDebug()` has been reached, the "Debug Info: ..." dialog pops up.

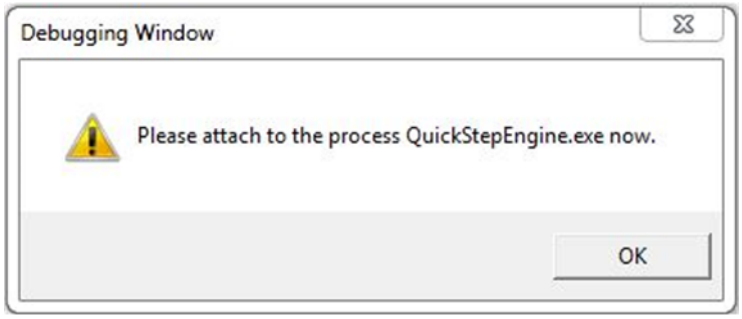


Figure 7-64: Debug Info ... dialog

6. Go to Visual Studio and select "Attach to Process..." from the "DEBUG" menu.

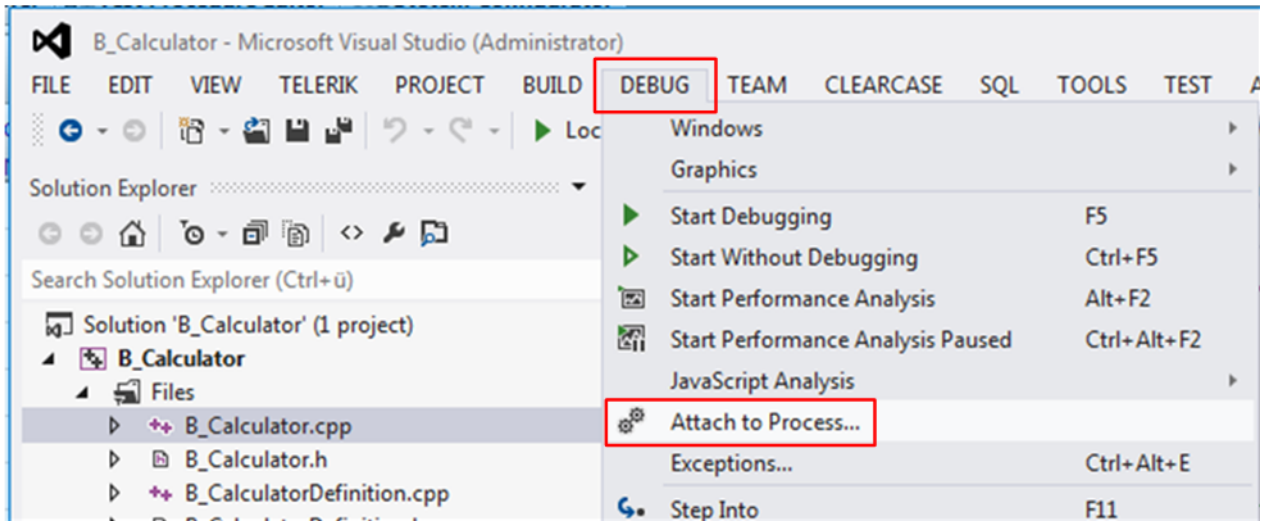


Figure 7-65: DEBUG > Attach to Process...

The "Attach to Process" dialog is opened.

7. Select the "QuickStepEngine.exe" process and click "Attach".

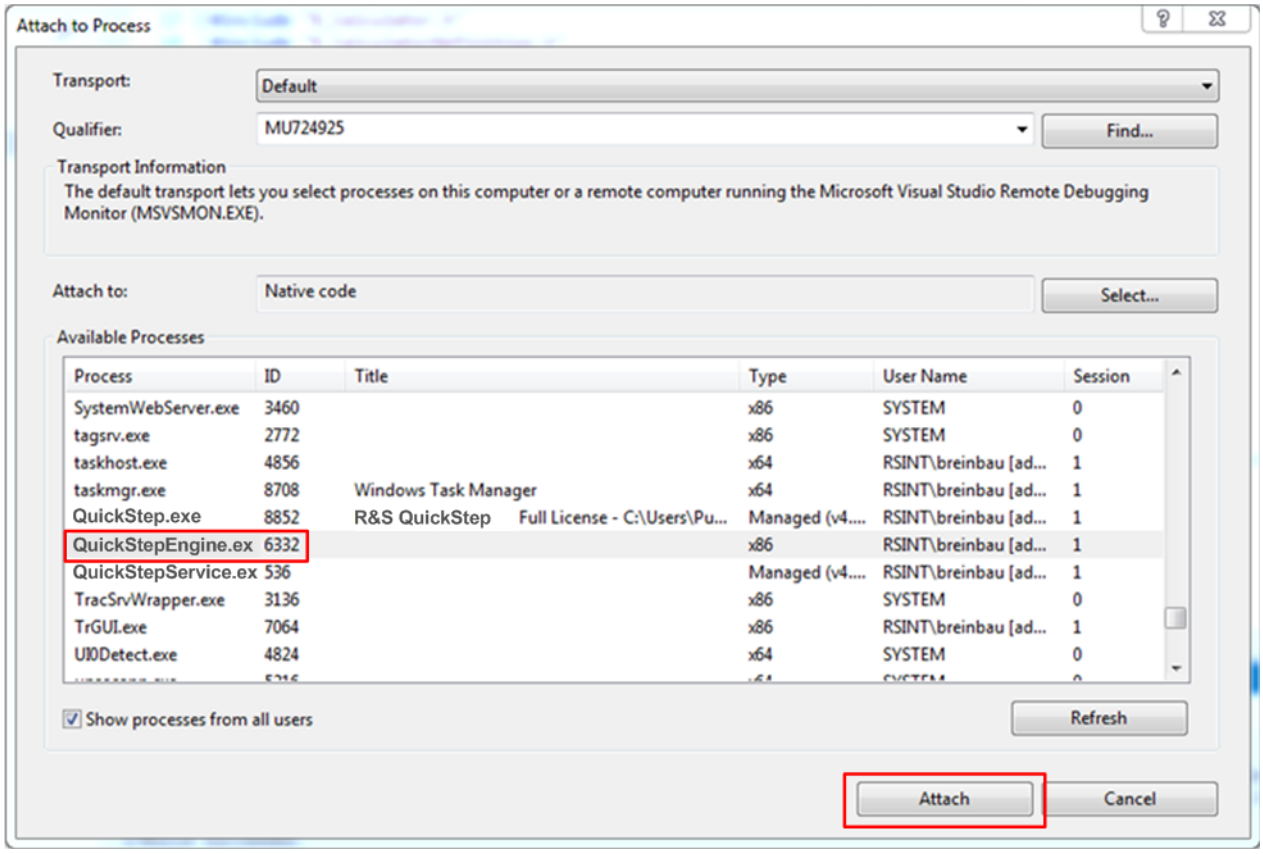


Figure 7-66: Attach QuickStepEngine to code

8. Back at the "Debug Info: ..." dialog, click "OK" to continue.

Result: The test plan execution stops at the Visual Studio breakpoint. You can apply the standard Visual Studio debugging facilities.

Make sure that the code section with the breakpoints is called only once during execution of the test plan. For example, the code section could be a function which is executed each test step. In this case set the number of repetitions to one. Alternatively, place `WaitDebug()` for example in the `B_Calculator` constructor which is executed only once or use `WaitDebugOnce()` instead. `WaitDebugOnce()` stops only when the line is hit for the first time.

7.2.8 Running QuickStep in Command Line Mode

Running QuickStep in command line mode may be helpful for special testing purposes.

1. Open a Command Prompt window.
2. Navigate to the folder `C:\Program Files\Rohde-Schwarz\QuickStep\Framework`.
3. Enter `QuickStepEngine.exe` to get information about the command options.
4. Enter `QuickStepEngine.exe <path to test plan>` to execute the specified test plan (<path to test plan> includes the name of the test plan).

   Options:
   - `<project path>` (appended after `<path to test plan>`): Only required if the test plan is not stored in the usual Project folder.
   - `--debug`: Test execution in debug mode.
   - `--single` or `--continue`: Test execution in Single Run mode or Continuous mode.
   - `--dcount <n>`: Number of DUTs for testing purposes. This flag is not available at the QuickStep GUI.

Related information: Chapter B, "Return Codes in Command Line Mode", on page 290

7.2.9 Re-Using an R&S Forum Script

This procedure shows by example how an existing R&S Forum script can be integrated and used in QuickStep. The adaptation of the script to QuickStep comprises creation and integration of QuickStep script parameters (PowerOffset whose value shall be set in the test plan table and Power as example), reporting to QuickStep and writing results in QuickStep result files.

Assumptions to keep things simple:

- The Forum script is available in the file system.
- The script controls one test instrument (R&S NRP as example).
- The Forum script already uses the PowerOffset and Power variables.
- Test plan and test procedure are already prepared and connected except for the missing script block for the Forum script in the test procedure.

myForumScript.i3e*

```
1  import math
2
3  # Print identity
4  NRP.ask("*IDN?")
5
6  # Set Power Offset
7  PowerOffset = 5
8  NRP.write("CORR:OFFS " + str(PowerOffset)
9  NRP.write("CORR:OFFS:STAT ON")
10
11 # Read measurement
12 NRP.write("INIT")
13 Power = NRP.ask("FETC?")
14
15 # Calculate dBm and write output
16 Power = float(Power[0])
17 Power = 10*math.log((1000*Power), 10)
18 print("Power: " + str(Power)[:7] + "dBm")
```

Figure 7-67: R&S Forum script to be re-used

Part 1: Importing the script and creating script parameters

1. Click "Import Forum Script" in the toolbar of the "Test Procedure Editor".
2. Browse to the folder containing the Forum script, select the script and click "Open".
   The "Add or Import Forum Script" dialog is opened.

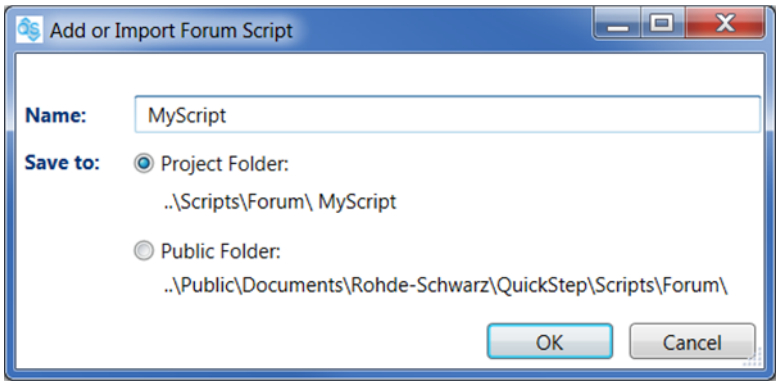


Figure 7-68: Add or Import Forum Script dialog

3. Accept or change the default "Name" and "Save to" selection and click "OK".
   (If "Public Folder" was selected: The script is accessible and editable from all QuickStep projects.)
   The "Edit Script Parameter" dialog is opened.
4. Add script parameters and their properties in the upper part of the dialog.

The parameter definitions are added in the script as shown in the lower part of the dialog.

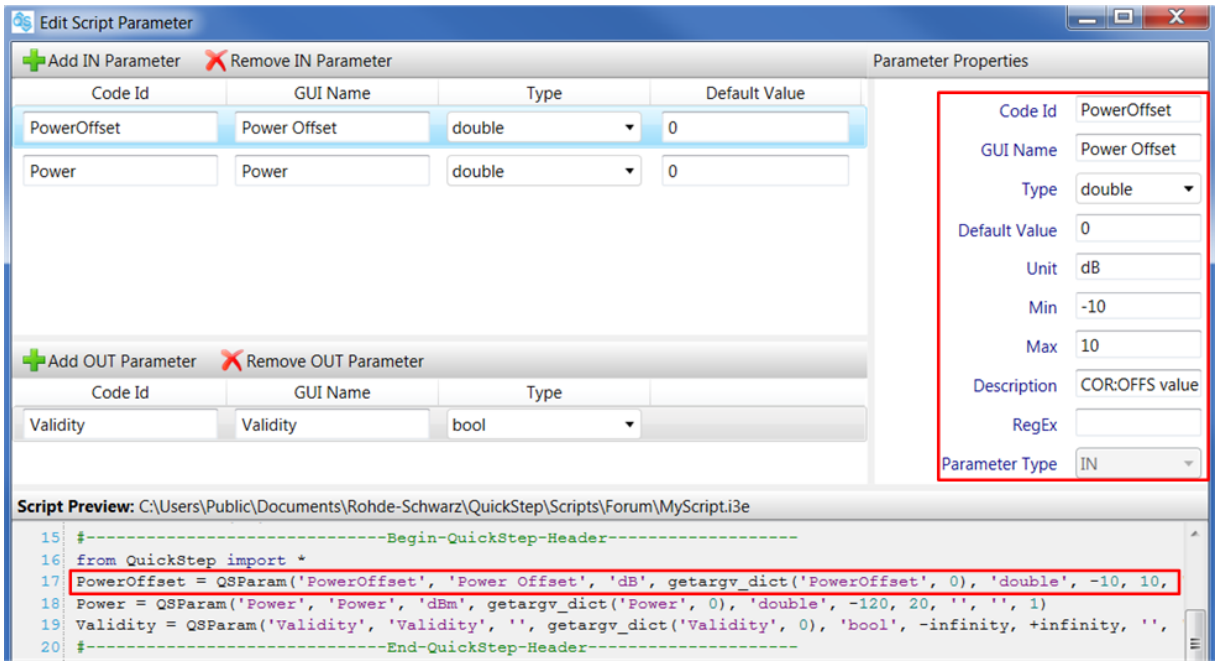


Figure 7-69: Edit Script Parameters dialog

5. Click "Save" to save the modified script.
   The import is executed, a Scriptblock for the script is now available in the Test Procedure Editor's "Library".

6. In the "Test Procedure Editor", select the "Test Procedure" tab in the main (middle) area and drag the new ScriptBlock from the "Library" into the "Test Procedure" area.
   The new parameters are displayed in the "Properties" area (with the script block selected in the main area).

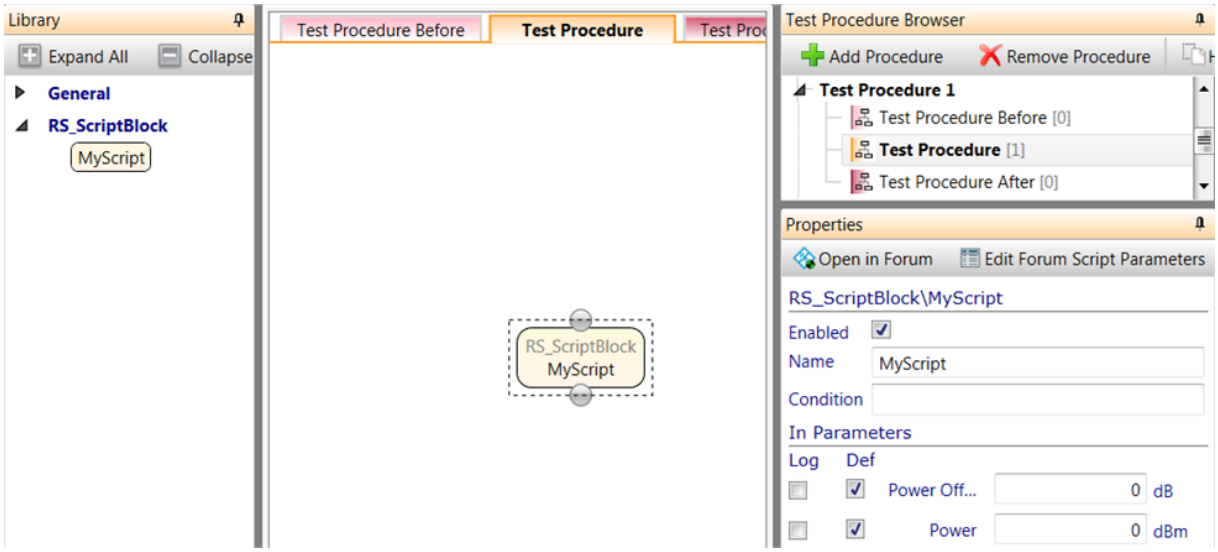


Figure 7-70: New ScriptBlock in the Test Procedure Editor

Part 2: Adapting the script in Forum

1. In the "Properties" pane in the Test Procedure Editor, click "Open in Forum". The script is opened in Forum.
2. In the Forum "EDITOR" pane, go to the `ForumDevices = { ... }` line and check if the VISA instrument to be controlled is available, here `NRP`. You can manually modify the list if desired.
   It is recommended that you set the VISA resource string in Forum ("Settings > Instruments..."), so that it will match with the VISA resource string in QuickStep (to be set later).
3. Check, add, modify the script lines where the QuickStep parameters are used, see the figure.
   Note: The elements of the PowerOffset variable handed over from QuickStep can be accessed with PowerOffset.value, PowerOffset.unit and so on.
4. Add the desired QuickStep functions and handle them with appropriate variables, see the figure.

MyScript.i3e*

```
1  import collections
2  import sys
3
4  # List of VISA_Resource parameters available in QuickStep, please add all used For
5  ForumDevices = {NRP}
6
7  # QuickStep Header - Do not add your own code
8  from QuickStep import *
9  PowerOffset = QSParameter(name='Power Offset', unit='dB', value=getargv(1, 0), typ
10 Power = QSParameter(name='Power', unit='dBm', value=getargv(2, 0), type='double',
11 # QuickStep Header - end
12
13 # ...
14
15 # Print identity
16 Idn = NRP.ask("*IDN?")
17 SendLogConsole(LogLevel.NORMAL, Idn)
18
19 # Set Power Offset
20 NRP.write("CORR:OFFS " + str(PowerOffset.value))
21 NRP.write("CORR:OFFS:STAT ON")
22
23 # Read measurement
24 NRP.write("INIT")
25 Power = NRP.ask("FETC?")
26
27 # Calculate dBm and write output
28 Power = float(Power[0])
29 Power = 10*math.log((1000*Power), 10)
30 SendAsyncLogResultDouble(ResultFileType.RESULT,'MyScript','MeasPow','dBm',4,Power)
```

Figure 7-71: Script with QuickStep parameters and functions

5. Save the edited file ([Ctrl + s]).

Part 3: Finalizing in QuickStep

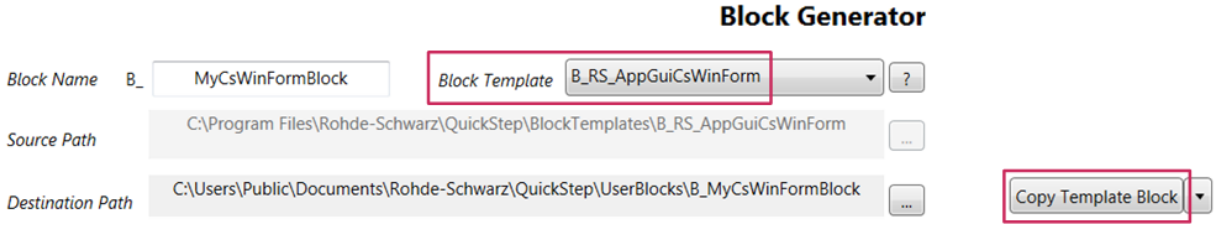
1. Back in the "Test Procedure Editor", click "Reload Block Library" at the toolbar to update the script block.
2. Set the value of the NRP VISA resource parameter.
3. Make sure that the check box for the Power Offset parameter is not ticked (consequently, this parameter will be visible in the test plan table).
4. Complete the test procedure according to your needs.
5. Go to the "Test Plan Editor" and click "Update Test Project" in the toolbar.
6. Set the parameter values in the test plan table according to your needs.
7. Click "Single Run" to execute the test.

Result: The main events are reported in the QuickStep "Log Viewer". The results (here the fetched power level) are stored in the standard QuickStep result files. If the QuickStep Log Level is set to VERBOSE, the output which is normally visible in the FORUM output window is also forwarded to the QuickStep "Log Viewer".

7.2.10 Creating and Integrating a User-Defined GUI

Summary: A user-defined dialog block is created with the Block Development Tool and via a template block. The dialog functionality is implemented in C# in Visual Studio. Eventually, a dialog block function is added in a QuickStep test procedure as any other user-defined block function.

1. Open the "R&S QuickStep Block Development Tool".
2. In the "Block Generator" section:
   - Enter a name for the block to be created.
   - Select "Copy Template Block" via the icon.
   - Select one of the available "Block Template"s. For example:
     – Select "B_RS_BlockingGuiCsWinForm" if you will adapt the dialog elements with Windows Forms and do not need to grant own resources (own thread) for the dialog (solution with low complexity).
     – Select "B_RS_NonBlockingGuiCsWpf" if you will implement a WPF dialog and if other block activities shall be available simultaneously during test execution (solution with high complexity).



3. Click the "Copy Template Block" button.
   A new dialog block with some default functions and default properties is created. The block is automatically loaded in the "Block Definition File Editor" area.
4. In the "Block Definition File Editor" add and configure provided functions according to your needs.
5. Click the "Save & Export Project" button.
6. Click the "Open with Explorer" button.

The opened Windows Explorer shows the files of the created project.

7. Open the block project (`.csproj` file) in Visual Studio.
8. Adapt the dialog elements and implement the functions previously defined in the Block Development Tool.
9. Compile the Visual Studio project.
   Now, the dialog block is available in the QuickStep "UserBlocks" folder.
10. In QuickStep, create or load the test project which shall get the dialog.
11. Go to the Test Procedure Editor, select "Blocks & Connectivity" in the Test Procedure Browser, expand "User Blocks" in the "Block Library" and drag the dialog block into the "Blocks & Connectivity" area.
12. Select the desired test execution phase in the Test Procedure Browser and drag a dialog block function into the test procedure area.
13. Update the test project in the Testplan Editor.

During test execution, the dialog is opened when the dialog block function is called according to its position in the test procedure.

8 Application Examples

Many application examples have predefined references to aliases of VISA resource strings in the "System Configurator". When a testplan is executed, QuickStep expects that the source of the VISA reference is available in the "VISA Instruments" table.

Hence, provide an entry for each referenced VISA alias in the "VISA Instruments" dialog (even if the alias is not used in the test procedure).

Otherwise, an error message is reported during test execution.

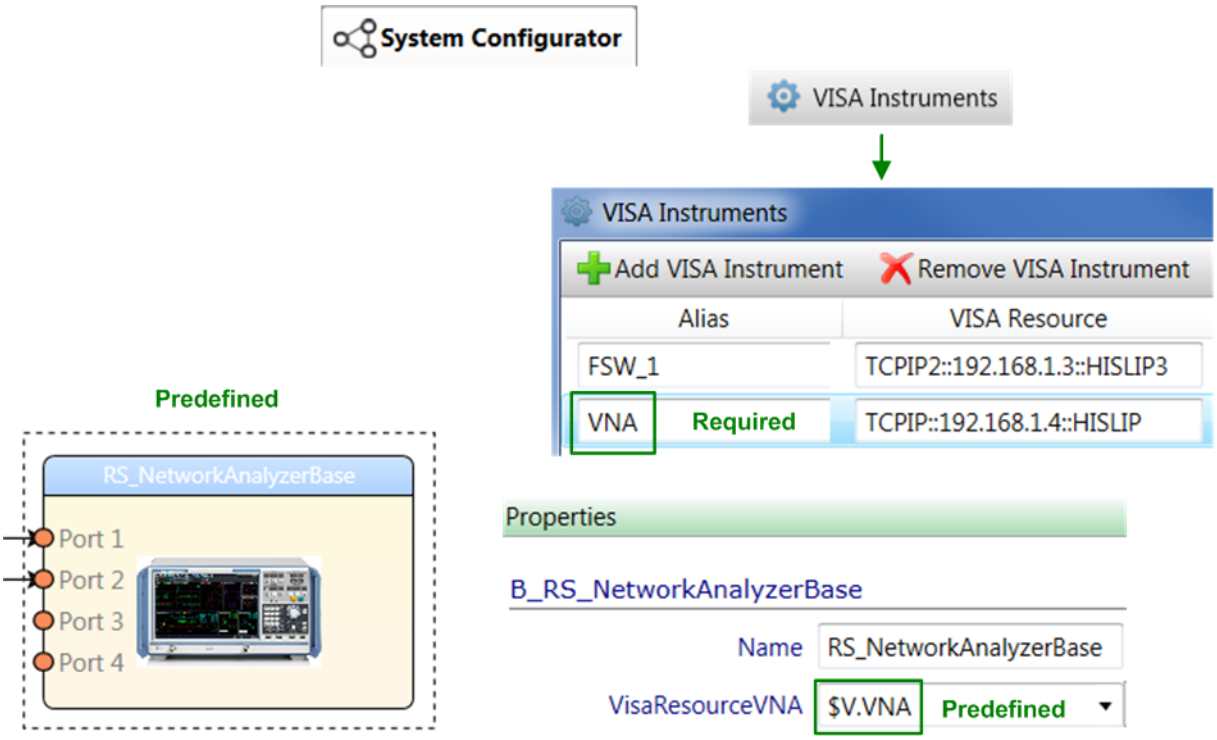


Figure 8-1: Predefined and required VISA information

8.1 Calculator

This example test project does not require any instruments connected to the PC. It is intended to get familiar with the basic elements of the QuickStep GUI. There are several test project variants accessible via `*.tpl` files:

- `Calculator.tpl`: Calculator example with C++ blocks

- `Calculator_withSingleLineSweeps.tpl`: C++ calculator example with several single-line sweeps within the test plan

8.1.1 Test Procedure

The test procedure contains two block functions which execute basic arithmetic operations (addition, subtraction, multiplication, division and power) of two input values. The block functions are executed in parallel as no dependency is set.

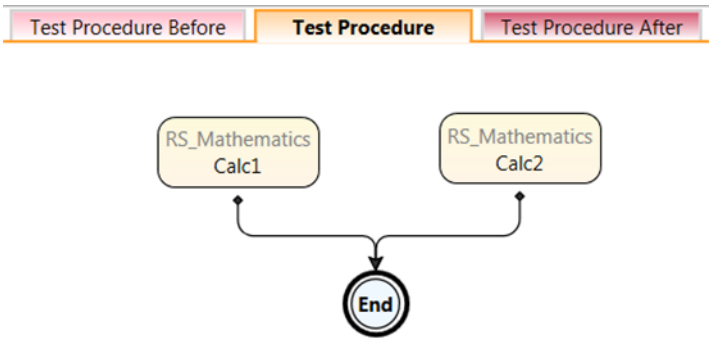
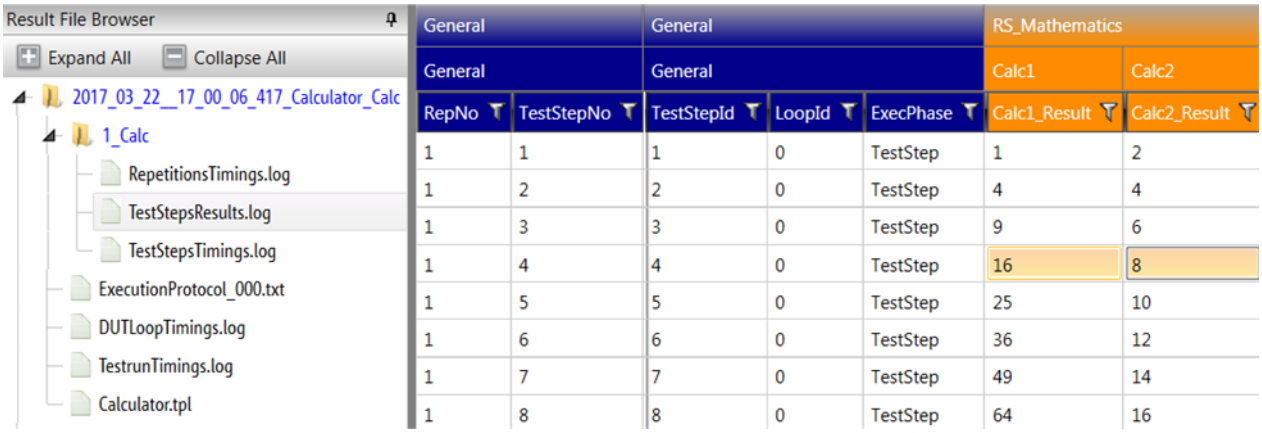


Figure 8-2: "Test Procedure" phase for the Calculator example project

In the test procedure shown in the figure, the block function "Calculation" is used but it has been renamed to "Calc1" and "Calc2".

8.1.2 Results

The following figures show the results of the `TestStepsResult.log` file in the central table and in the diagram. Note that two table cells in the "Calc1_Result" and "Calc2_Result" columns have been selected to load data into the diagram.



General		General			RS_Mathematics	
General		General			Calc1	Calc2
RepNo	TestStepNo	TestStepId	LoopId	ExecPhase	Calc1_Result	Calc2_Result
1	1	1	0	TestStep	1	2
1	2	2	0	TestStep	4	4
1	3	3	0	TestStep	9	6
1	4	4	0	TestStep	16	8
1	5	5	0	TestStep	25	10
1	6	6	0	TestStep	36	12
1	7	7	0	TestStep	49	14
1	8	8	0	TestStep	64	16

Figure 8-3: Results table for the calculator example project

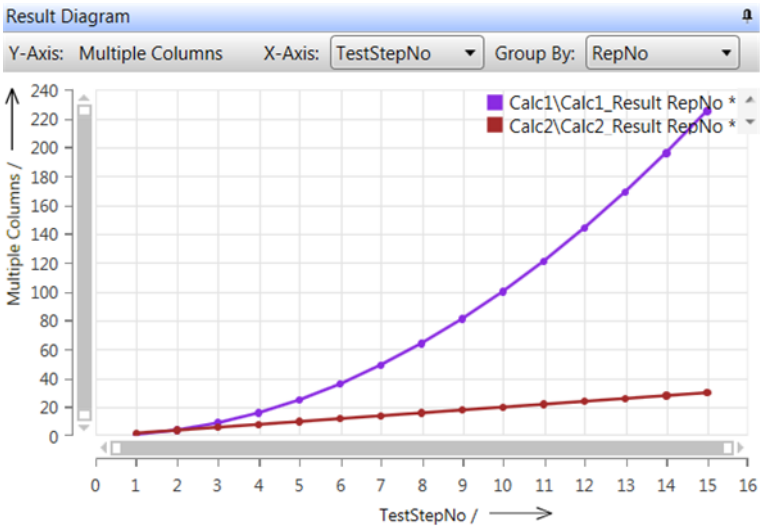


Figure 8-4: Diagram representation of calculator results

8.2 Control Statements

This example demonstrates how to use the control statements If and Or and a loop in a test procedure. No test instrument is required.

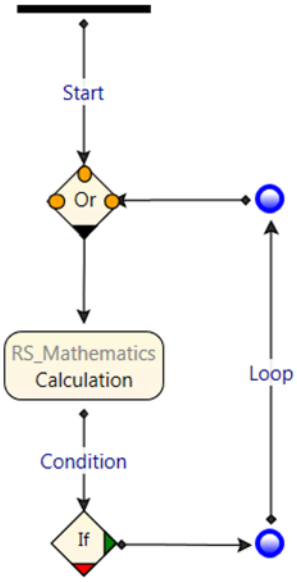


Figure 8-5: Test procedure with control statements and a loop

8.3 DC PowerSupplies

The "DC_PowerSupplies" example test project is intended to show the basic usage of a DC power supply.

8.3.1 Test Setup and Usage of Components

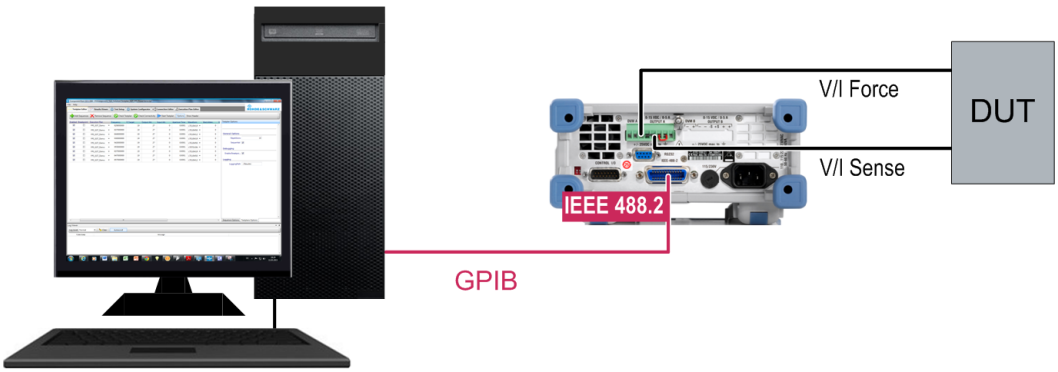


Figure 8-6: Test setup for the DC PowerSupplies example

Characteristics:

- The power supply feeds the DUT with DC input power.
- The power supply measures the DC output current of the DUT.

- The PC running QuickStep is connected to the power supply via GPIB or LAN.

8.3.2 Test Procedure

The actual testing takes place in the "Test Procedure" phase.

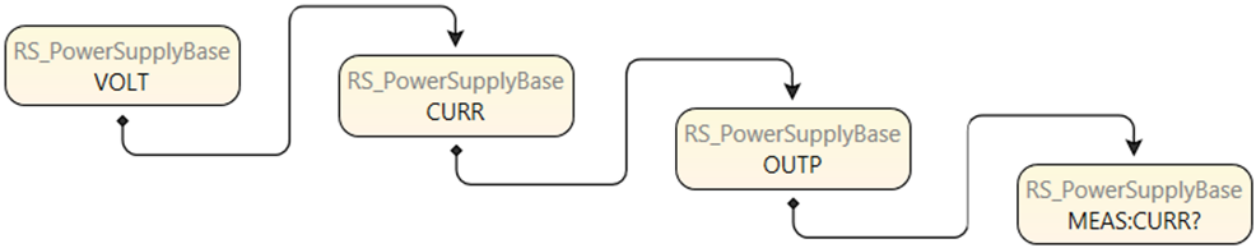


Figure 8-7: "Test Procedure" phase of DC PowerSupplies

The voltage and current for feeding the DUT is defined. The resulting output current of the DUT is measured.

8.4 Dual Instance Power Sensors

This example test project evaluates the power measurements of two connected power sensors.

8.4.1 Test Setup and Usage of Components

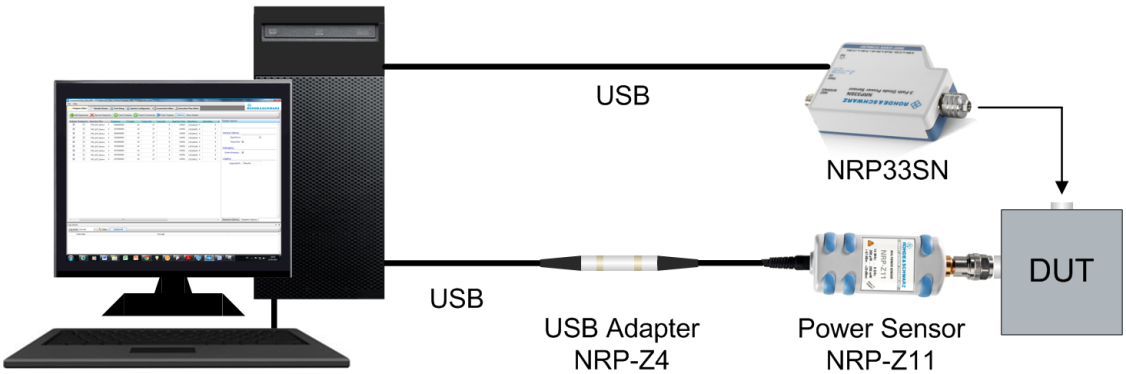


Figure 8-8: Test setup for dual instance power sensors

Characteristics:

- The power sensors are connected to the DUT via RF connectors and measure the RF power of the DUT.
- The PC with QuickStep is USB-connected to the power sensor (in case of an R&S NRP-Z sensor with the help of a USB adapter). QuickStep controls the power sensors and reads its measurement results.

8.4.2 Test Procedure

Two instances of the RS_PowerSensorBase block are created in the "Blocks & Connectivity" section, one for each power sensor. This allows for independent and parallel processing of the block functions for the power sensors (two threads).

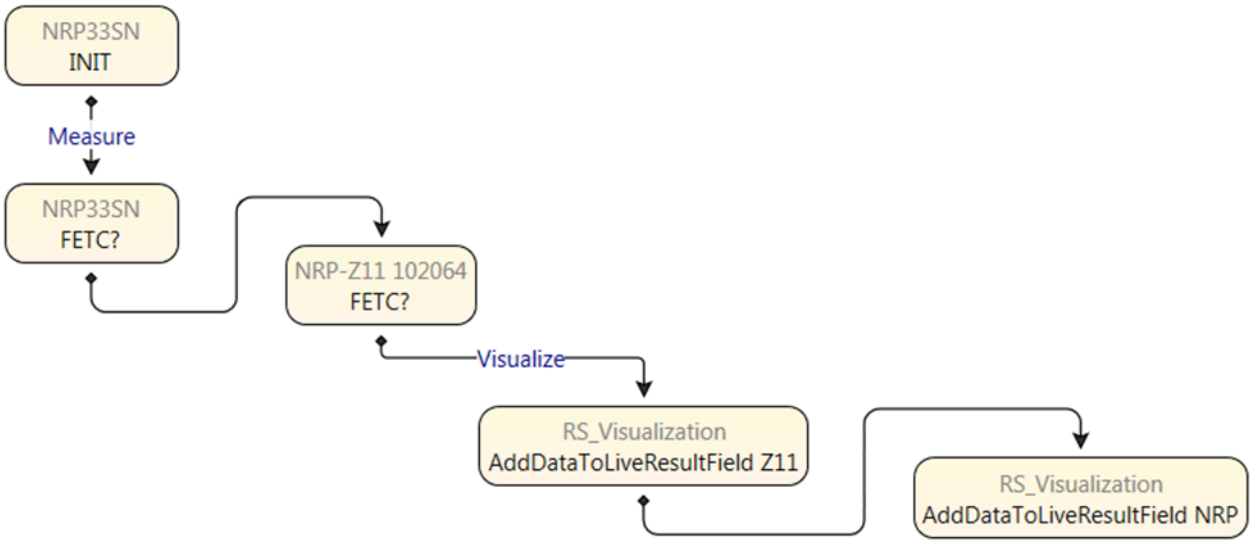


Figure 8-9: "Test Procedure" phase for power sensor measurements

The "FETC?" function fetches the last valid power measurement result from the power sensor.

The provided test procedure is a template combining the control of both an NRP-Z11 and an NRP33SN power sensor. If only one power sensor is used, remove the power sensor block which is not used from the test procedure.

8.5 Forum Scripting

This example applies a Forum script which controls a power sensor measurement, calculates statistical measures and provides log reports.

The test setup includes an NRP power sensor.

The Forum script to be executed is available under C:\Users\Public\Documents \Rohde-Schwarz\QuickStep\Examples\Forum_Scripting\Scripts.

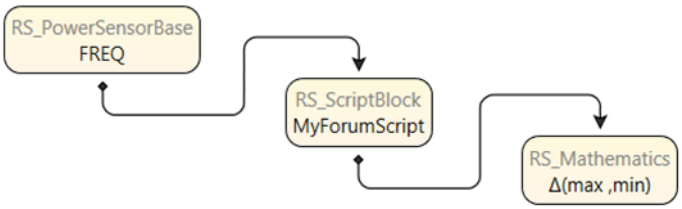


Figure 8-10: Test procedure applying a Forum script

8.6 Matlab Scripting

This example applies a Matlab script which simulates a modulator, noisy channel and a demodulator. QuickStep evaluates the modulation performance under increasing signal to noise ratios.

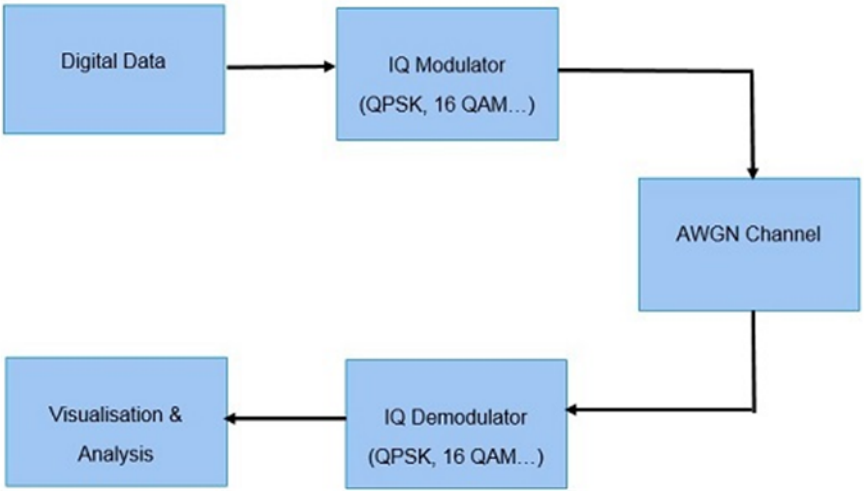


Figure 8-11: Functional parts in the Matlab script

The Matlab script to be executed is available under C:\Users\Public\Documents \Rohde-Schwarz\QuickStep\Examples\Matlab_Scripting\Scripts.

Results

The results of the simulation are visualized in a series of plots.

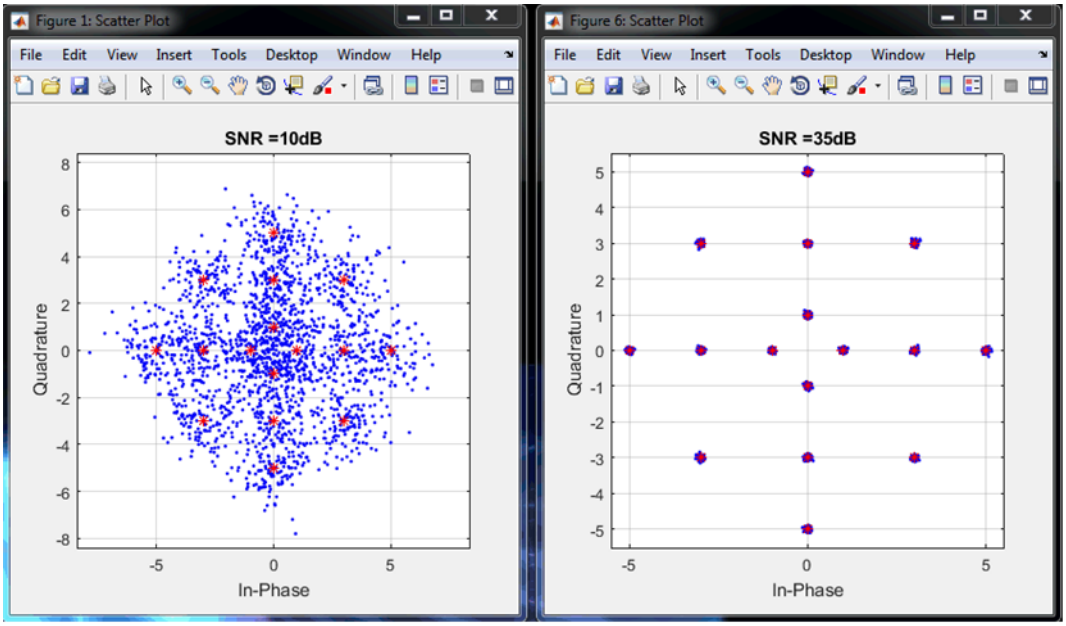


Figure 8-12: Results of the test execution (selection)

8.7 Network Analyzer

The "Network_Analyzer" example test project controls a vector network analyzer for measuring the S parameters S11, S12, S21 and S22. The S parameters (scattering parameters) characterize the transmission and reflection behavior in high-frequency networks.

8.7.1 Test Setup and Usage of Components

Port 1 and port 2 of the network analyzer are connected with the DUT which is a frequency selective (lowpass) element. In case of a 4-port network analyzer, a second similar DUT is connected between port 3 and port 4.

Figure 8-13: Test setup with a network analyzer

At the network analyzer, two new channels with four traces each are created for measuring all S parameters.

At QuickStep on the PC, the frequency of the created channel can be set. All traces are displayed in the same window.

8.7.2 Test Procedure

For demonstration purposes, three different procedures are used.

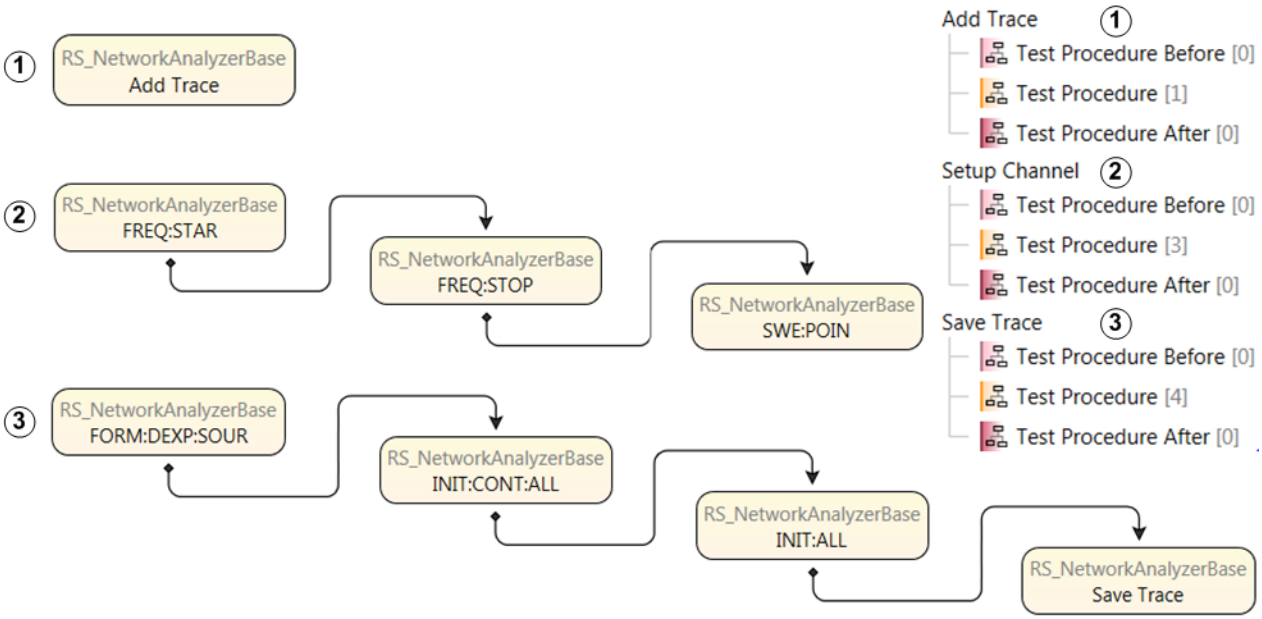


Figure 8-14: "Test Procedure" phase of the network analyzer test

- Add Trace procedure: The "Add Trace" function defines traces for S parameter measurements and connects them to channels and windows (this is done in test step parameters within the Testplan Editor, one trace per test step).
- Setup Channel procedure: The "FREQ:STAR", "FREQ:STOP" and "SWE:POIN" functions define the sweep frequency range and resolution for the channels.
- Save Trace procedure: The functions do final preparations (for example defining continuous sweep) and execute the frequency sweep and save S parameter measurements in traces as defined in the first procedure.

8.7.3 Test Plan, Results

In the "Testplan Editor", the "Add Trace" section contains the main settings: The parameters define per test step which S parameter measurement is executed over which channel, which trace and window belongs to the measurement and how the raw array of sweep results for one S parameter measurement are weighted.

When executing the test, the traces of the channels are saved in one trace file for each channel.

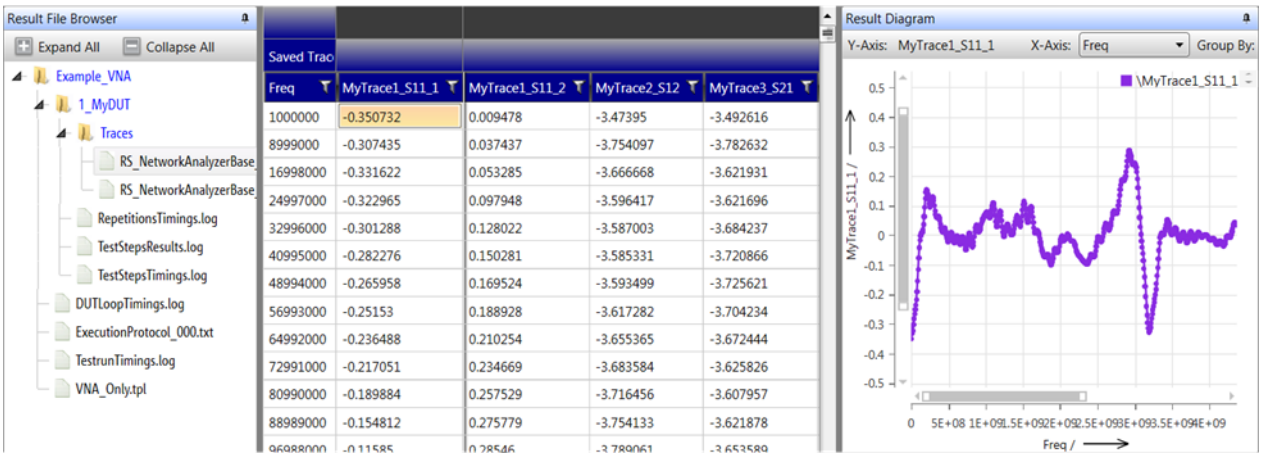


Figure 8-15: Results of network analysis

8.8 OSP Switching Unit

8.8.1 Test Setup and Usage of Components

Characteristics:

- The relays of the R&S OSP (Open Switching and Control Platform) are connected to different DUTs or RF devices to switch between.
- Relays can be set, individually or together as path, and read back and compared to a user's expectation.

8.8.2 Test Procedure

The actual testing takes place in the "Test Procedure" phase.

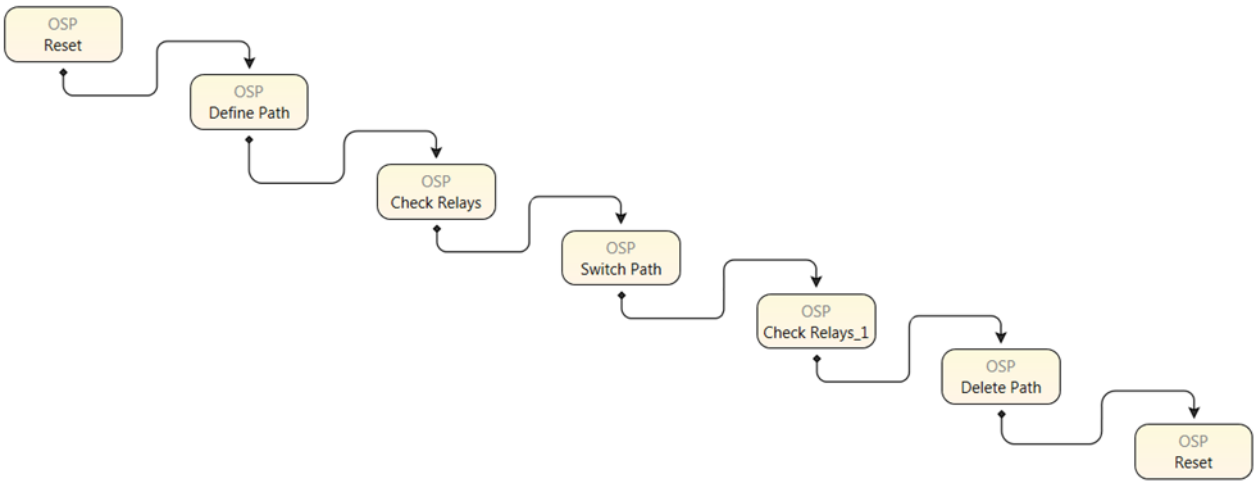


Figure 8-16: "Test Procedure" phase of OSP Switching Unit

- A reset is done to get a defined starting state.
- A path, that is a set of relays and their state, is defined. It is saved under a certain name for further usage.
- The state of the relays is read and compared to the user's expectation.
- The relays are switched according to the previously defined path.
- The state of the relays is read back and compared for verification.
- The defined path is deleted and the R&S OSP reset.

Note:

A list of relays and their state is declared as Channel List.

Syntax: (@FxxAyy(ccnn)) with:

Fxx: Instrument name, xx = 01, ..., 09

Ayy: Module name, yy = 11, 12, 13

cc: Condition of the relay to be set, cc = 00, 01, 02, ..., 06

nn: Relay/channel number within a module, nn = 01, 02, 03, ..., 16

8.8.3 Results

The OSP does not perform measurements. No direct test results are received. Execution times and execution protocol can be verified.

8.9 Reporting

This example illustrates how results can be collected in a report. The report includes a subreport.

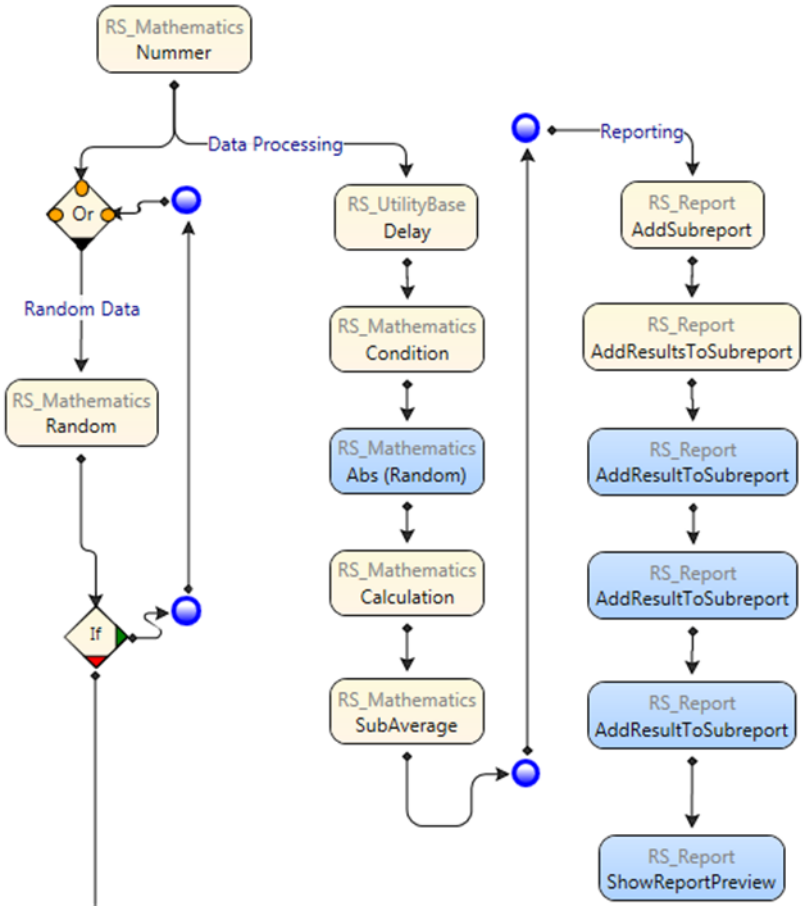


Figure 8-17: Test procedure for reporting

8.10 RTO Oscilloscope

The "RTO_Oscilloscope" example test projects selects a channel, records a measurement trace and fetches that trace.

8.10.1 Test Setup

The R&S RTO is LAN-connected with the QuickStep PC and RF-connected to the device under test (the test plan can even be run without connecting a device; in this case, noise is measured).

8.10.2 Test Procedure

The actual testing takes place in the "TestProcedure" phase.

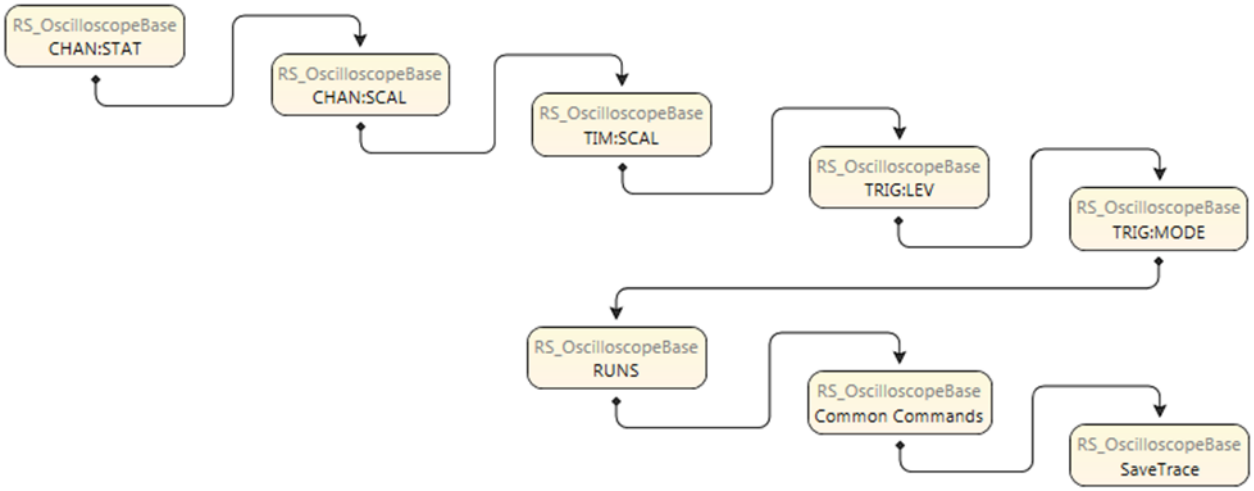


Figure 8-18: "Test Procedure" phase of RTO_Oscilloscope

- The measurement channel is selected and activated.
- The scaling in y direction and in the time direction is configured.
- The trigger is configured.
- An oscilloscope sweep is performed and the trace recorded.
- The trace file is saved.

8.11 Signal Analyzer

This example test projects contains two test plans for ACLR and Peak Hold measurements with an R&S FSW signal analyzer. Reporting and visualization is also included.

8.11.1 Test Procedure

The actual testing takes place in the "TestProcedure" phase.

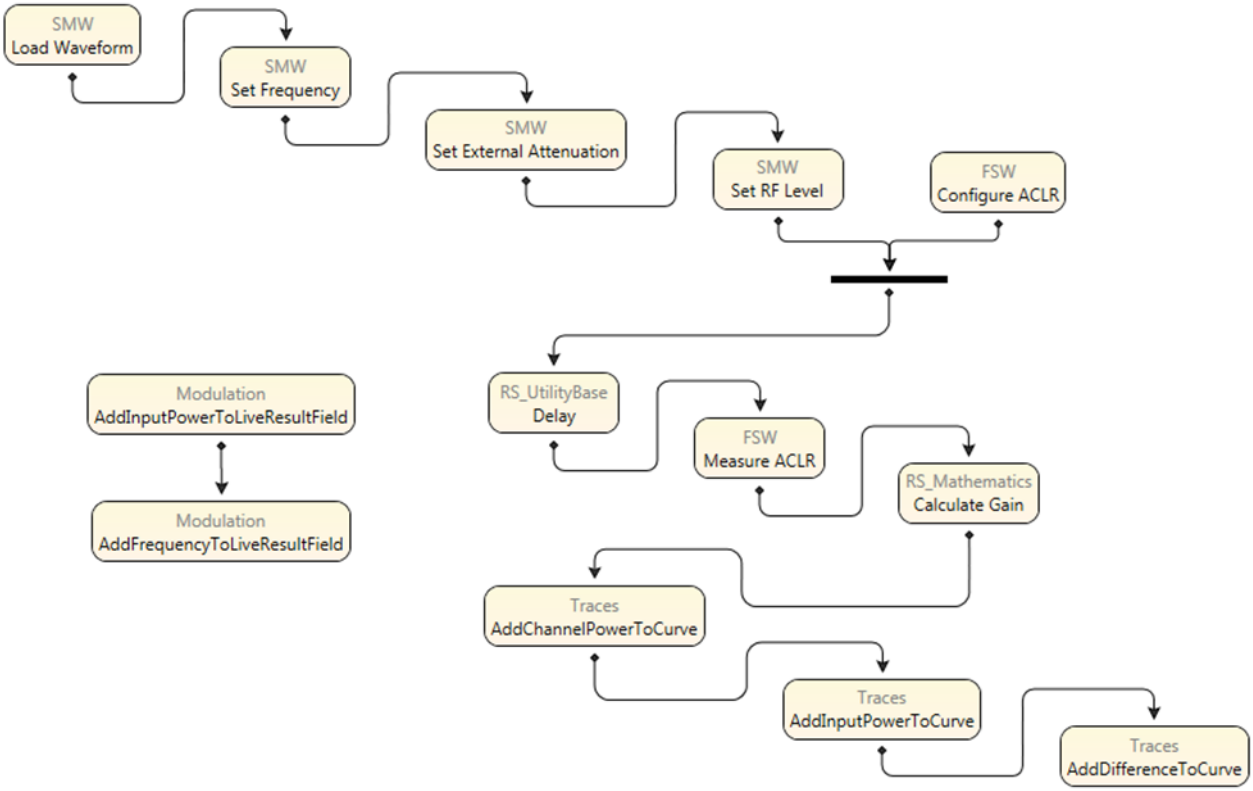


Figure 8-19: "Test Procedure" phase for ACLR measurement

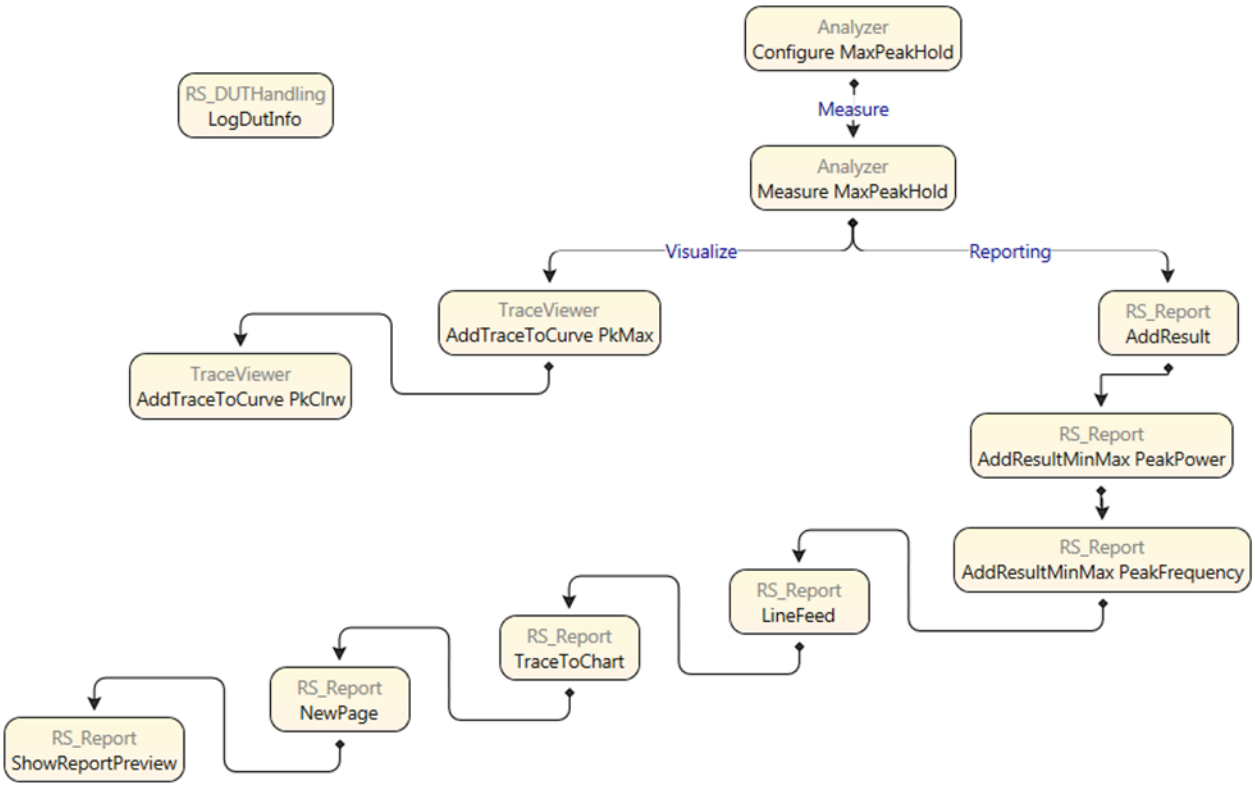


Figure 8-20: "Test Procedure" phase for the peak hold measurement

8.12 Visualization

This example creates a curve and a histogram from simulated data or results from a power sensor measurement.

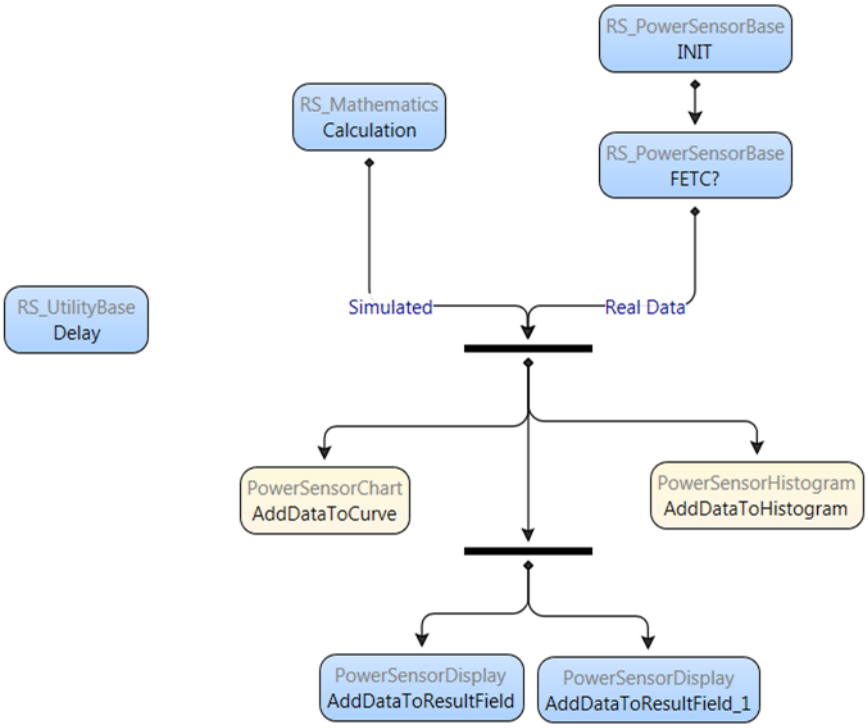


Figure 8-21: Test procedure with visualization block functions

8.13 Positioner Block Solution

QuickStep provides the positioner block solution to control an antenna positioner. The solution can either be used within QuickStep test procedures in the usual way or without using QuickStep. In the latter case, an application program is needed to call the block functions which are provided by DLLs, see "Application program" on page 203 for an example. This program has to be set up manually (ask for customer support if you need help).

The positioner block solution consists of the following libraries:

- `B_RS_Positioner.dll`: Contains the functions to control the antenna positioner
- `B_RS_PositionerRT.dll`: Contains functions related to the runtime environment. These functions mainly provide logging, tracing, VISA connection and timer functionality. Actually, the `B_RS_PositionerRT.dll` overrides basic environment functions which are contained in the package as embedded resources.

- `Common_cs.dll` and `CommonBaseCS.dll`: Contain environment functions needed for managing and executing block functions.

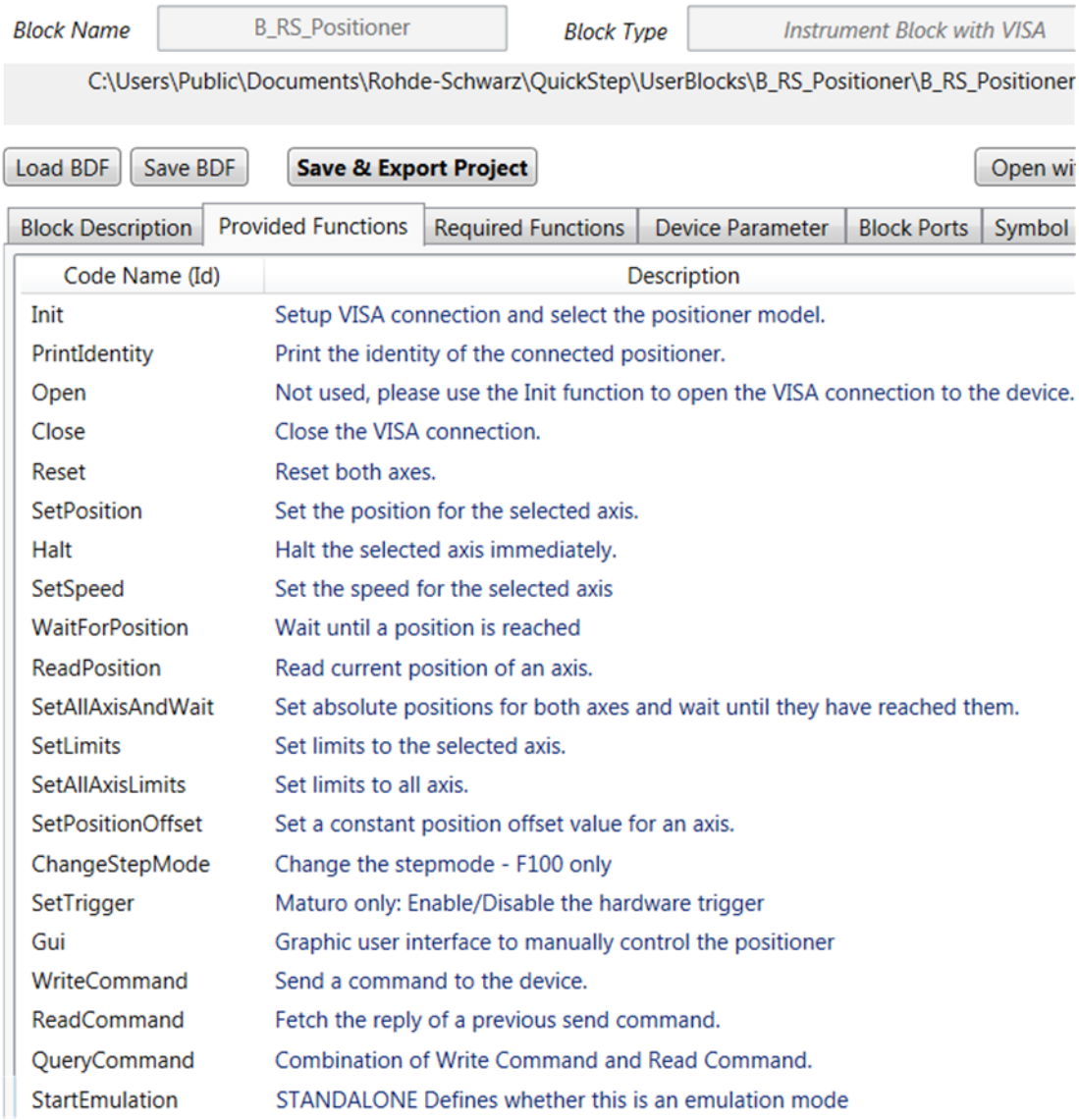
The following positioner types are supported:

- ATS-CCP1 antenna positioner with turntable and one antenna boom
- F100, F200 pan/tilt antenna positioner

A simulation mode is also available for executing tests without hardware.

B_RS_Positioner block functions

The figure shows the available block functions in QuickStep's Block Development Tool. See Chapter 9.3.1.9, "RS_Positioner", on page 268 for a short description of the block functions.



Code Name (Id)	Description
Init	Setup VISA connection and select the positioner model.
PrintIdentity	Print the identity of the connected positioner.
Open	Not used, please use the Init function to open the VISA connection to the device.
Close	Close the VISA connection.
Reset	Reset both axes.
SetPosition	Set the position for the selected axis.
Halt	Halt the selected axis immediately.
SetSpeed	Set the speed for the selected axis
WaitForPosition	Wait until a position is reached
ReadPosition	Read current position of an axis.
SetAllAxisAndWait	Set absolute positions for both axes and wait until they have reached them.
SetLimits	Set limits to the selected axis.
SetAllAxisLimits	Set limits to all axis.
SetPositionOffset	Set a constant position offset value for an axis.
ChangeStepMode	Change the stepmode - F100 only
SetTrigger	Maturo only: Enable/Disable the hardware trigger
Gui	Graphic user interface to manually control the positioner
WriteCommand	Send a command to the device.
ReadCommand	Fetch the reply of a previous send command.
QueryCommand	Combination of Write Command and Read Command.
StartEmulation	STANDALONE Defines whether this is an emulation mode

Figure 8-22: B_RS_Positioner block functions

Application program

The figure shows a minimum application program to control the positioner by calling the positioner block functions. Casting with Int is applied due to used enum type.

```
using System;

using B_RS_Positioner;
using B_RS_PositionerRT
using Common_cs;

namespace DirectDLLPositionerExample
{
    class Program
    {
        public static void LoggingCallback(object c, System.EventArgs e)
        {
            Console.WriteLine(((CustomerRuntimeBase.LogEventArgs)e).LogText);
        }

        static void Main(string[] args)
        {
            // create an instance to the class
            B_RS_Positioner positioner = new B_RS_Positioner("PositionerID", "Simulation", RS_Common.eLogLevel.NORMAL,
                                                                IntPtr.Zero, RS_Common.BlockExecutionMode.DllCall);
            //Add listener
            positioner.AddLoggingEventHandler(new EventHandler(LoggingCallback));

            // operate the device in real mode or emulation mode
            positioner.StartEmulation(isEmu);

            // initiate all the required parameters
            positioner.Init((int)DeviceType.ATSCCP1, "TCPIP::172.16.21.50::200::SOCKET",
                                                (int)StepModeMATURO.degreeSec, 72, 20, -15, 375, -10, 165);
            // read the identification of the device
            Console.WriteLine(positioner.GetIdentity().replyIdentity);

            // set speed
            positioner.SetSpeed((int)AxisType.azimuth, 10, (int)Mode.absolute);
            positioner.SetSpeed((int)AxisType.elevation, 10, (int)Mode.absolute);

            // set azimuth angle and wait till it is done
            positioner.SetPosition((int)AxisType.azimuth, 45, (int)Mode.absolute);
            positioner.WaitForPosition();

            // reset to the origin point
            positioner.SetAllAxisAndWait(0, 0, 100);

            // close the VISA connection to the positioner
            positioner.Close();

            Console.WriteLine("Press key to finish.");
            Console.Read();
        }
    }
}
```

Figure 8-23: DirectDLLPositionerExample

Such an application program can be set up with various programming or scripting languages. In case of a Matlab script, use an event handler as described in "Direct DLL call with a Matlab script" on page 144.

9 Reference

The sections in this chapter are ordered according to the structure and tabs of the graphical user interface.

9.1 GUI

9.1.1 Top Menu Bar

The top menu bar covers file handling and product information such as access to the user manual.

Table 9-1: Top menu bar

Item	Options
"File"	• "New Test Project": Opens a new test project with default settings. • "Open Test Project": For loading a `*.tpl` test plan file. It includes not only test steps table and test procedures, but also block connections, system configuration and other information as available. • "Save Test Project": For saving the current project in a `*.tpl` file. • "Save Test Plan As...:" Opens a Windows Explorer for selecting a directory and a filename under which the test plan is stored. • "Create Testplan Execution Shortcut": Select "Single Run" or "Continuous Run" in the submenu to create a shortcut on the desktop for executing the currently loaded testplan. The shortcut allows you to execute a testplan without starting QuickStep. • "Import Project": Copies the content of a zip file into the QuickStep directory for projects. The folder structure of the zipped content is kept. If configuration files, result files or user-defined blocks shall be included in the import, these files have to arranged in the ususal directory structures for projects (see the provided example projects). Use `Projects` as root node in the zipped test project. It is recommended to import projects which have originally been exported. In this way, the folder structure is automatically correct. • "Export Project": Collects all files required for the current project, packs them into a zip container and stores them at a selectable location. The required elements (blocks, project files, code, results) are derived from the currently loaded `.tpl` file. A popup window allows to select if results or user block projects (code) shall be included. See Figure 9-1. Note: 3rd party dll's outside of the user directory cannot be detected and are not included in the export. • "Recent Projects": Lists the recently used `*.tpl` files and allows to select one of them. • "Exit": Closes QuickStep. If the last modifications have not been saved, a save dialog is provided.
"Help"	• "QuickStep Help" • "Getting Started Manual" • "User Manual": For accessing the user manual. • "User Training Manual" • "Developer Training C++ Manual" • "Developer Training C# Manual" • "End-User License Agreement" • "Open Source Acknowledgment" • "About...": Provides version information and a short description of the application.
"Window"	• "Reset Window Layout": The panes (if undocked: subwindows) are reset to their default positions and sizes within the full size main window. In this way you can easily get back undocked subwindows in the main window.

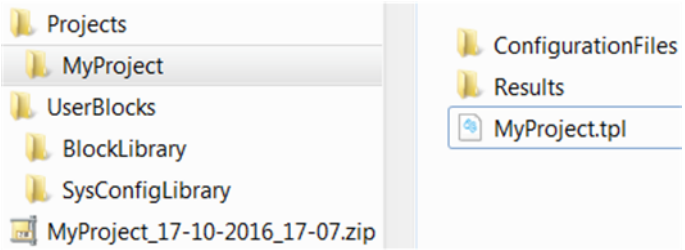


Figure 9-1: Exported project as zip file and unzipped in Explorer

9.1.2 General GUI Features

9.1.2.1 Arranging Panes/Subwindows

If you click a QuickStep pane's title bar or tab (if available) and then drag the pane, it is undocked and becomes an individual window. You can drag the undocked subwindow anywhere, even outside the QuickStep main window, or you can redock the subwindow at predefined positions in the QuickStep window. Arrow icons indicate predefined docking positions. Move the mouse pointer (still dragging the window) over an arrow icon to get a yellow marked preview of the target area. Release the mouse pointer to anchor the window as shown in the preview. See the figures.

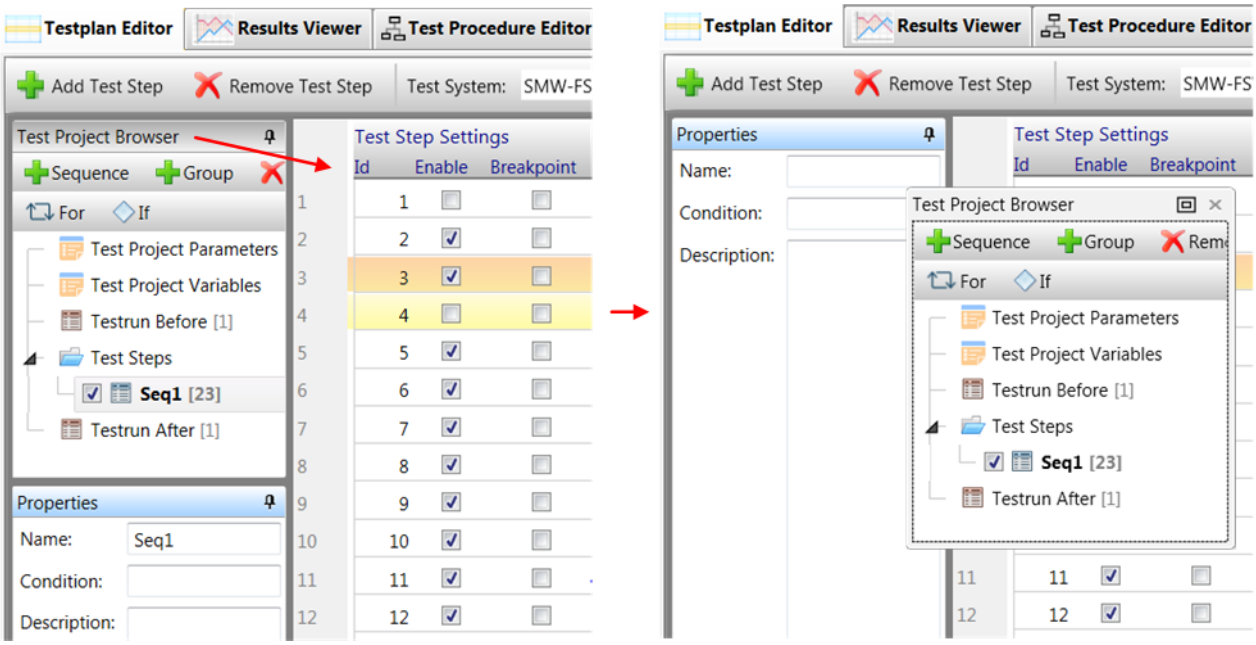


Figure 9-2: Undocking a pane by dragging its title bar

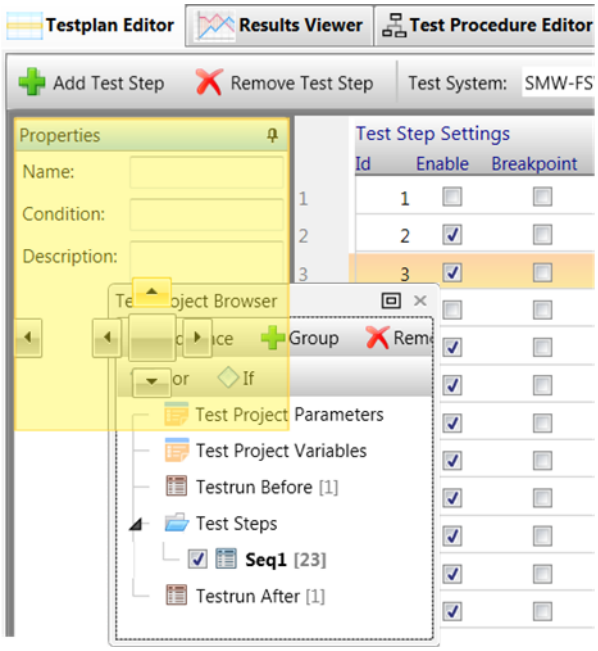


Figure 9-3: Preview of docking position

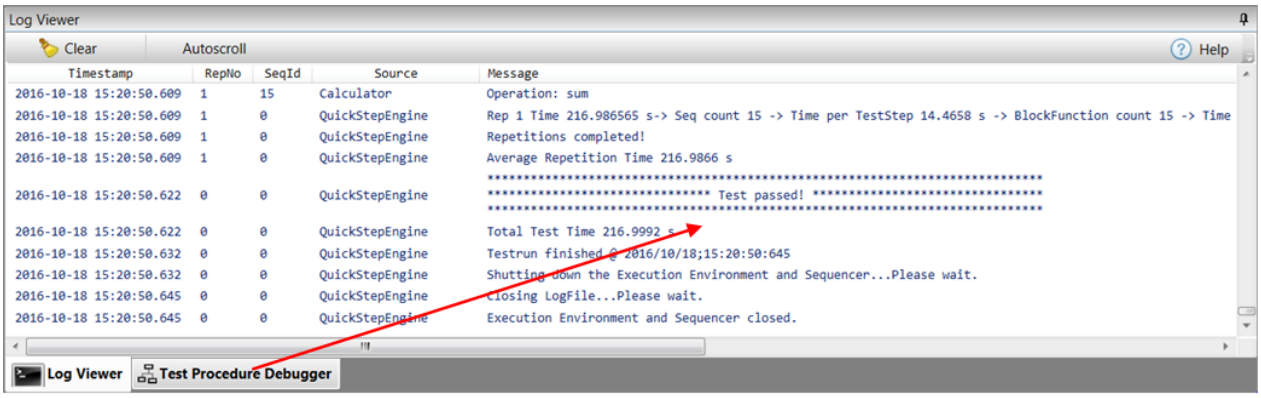


Figure 9-4: Undocking a pane by dragging its tab

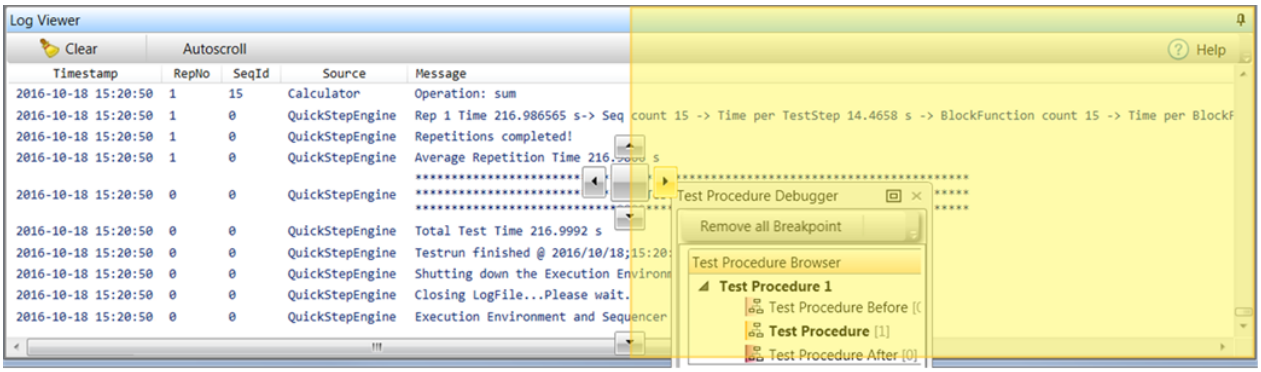


Figure 9-5: Preview of docking position

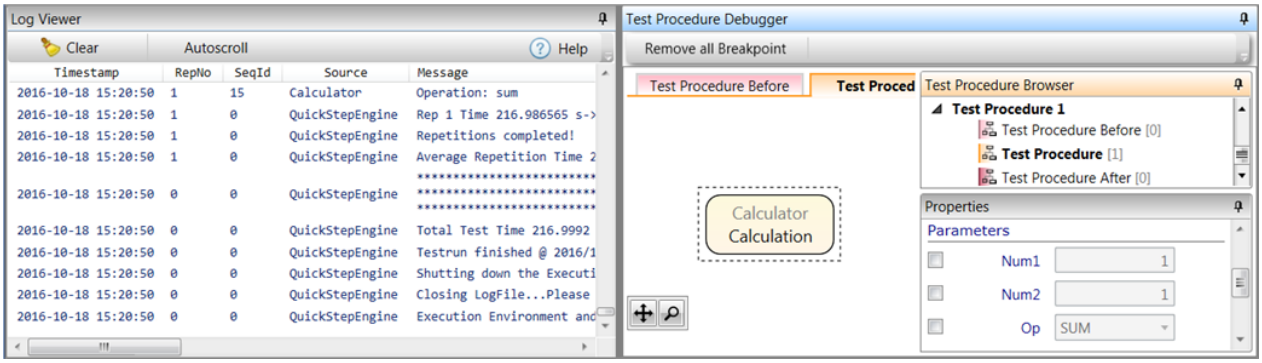


Figure 9-6: Resulting docking position

QuickStep stores the current positions and sizes of the QuickStep panes and undocked subwindows. If you close QuickStep and start the program again, you get back the last arrangement of panes and undocked subwindows.

9.1.2.2 Assisted Editing of Parameter Values

Set Reference dialog

The "Set Reference" dialog assists in setting values by reference. When clicking "OK" in the dialog, the dialog entries are converted into a reference string which appears as value for the referencing parameter (the user might enter such a reference string manually in the parameter value field). The dialog is available for parameters in various contexts, for example for "Test Step Parameters" within the "Testplan Editor" or "Properties" parameters within the "Test Procedure Editor". It is opened when a parameter input field is right-clicked and "Set Reference" is selected.

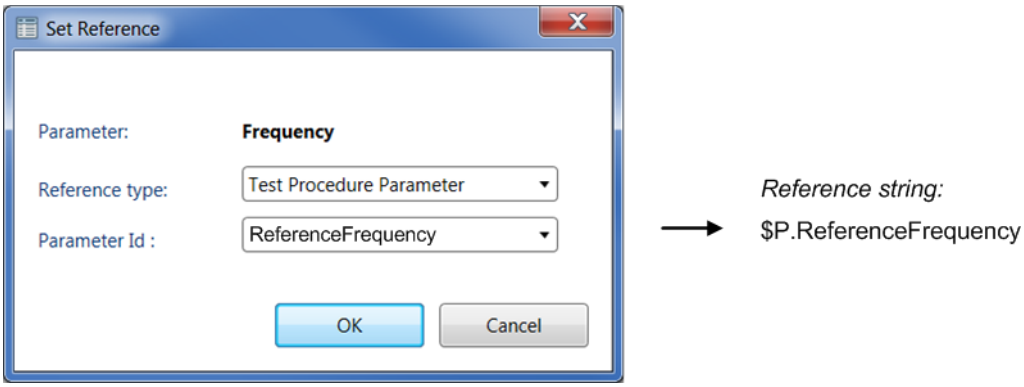


Figure 9-7: Set Reference dialog

- "Parameter" repeats the name of the parameter which shall get its value by reference.
- "Reference type" determines the area from which the value is fetched by reference (for example the Test Procedure area which is described by the *$P.* prefix in the reference string).

  Selectable values:
  - "None": No reference is defined.
  - "Mapping Parameter": The referenced parameter is located in the system configuration area. Such parameters are only available if they have been created and mapped by the "Mapping Table" dialog within the "Testplan Editor" before.
  - "Result Reference": The referenced parameter is a result of executed test steps. This selection extends the dialog with additional choices, see below.
  - "Test Project Parameter": The referenced parameter is available under "Test Plan Editor > Test Project Browser > Test Project Parameters" and with Project Parameter Type "Constant".
  - "Test Project Variable (Global)": The referenced parameter is available under "Test Plan Editor > Test Project Browser > Test Project Parameters" and with Project Parameter Type "Variable"
  - "Test Procedure Parameter": The referenced parameter is available in the "Properties" area of the "Test Procedure Editor".
- "Parameter Id" lists all parameters which can be selected for referencing in the current area.

The following figure shows the additional choices in the "Set Reference" dialog if "Result Reference" has been selected for the "Reference type". Corresponding elements are indicated with the same numbers.

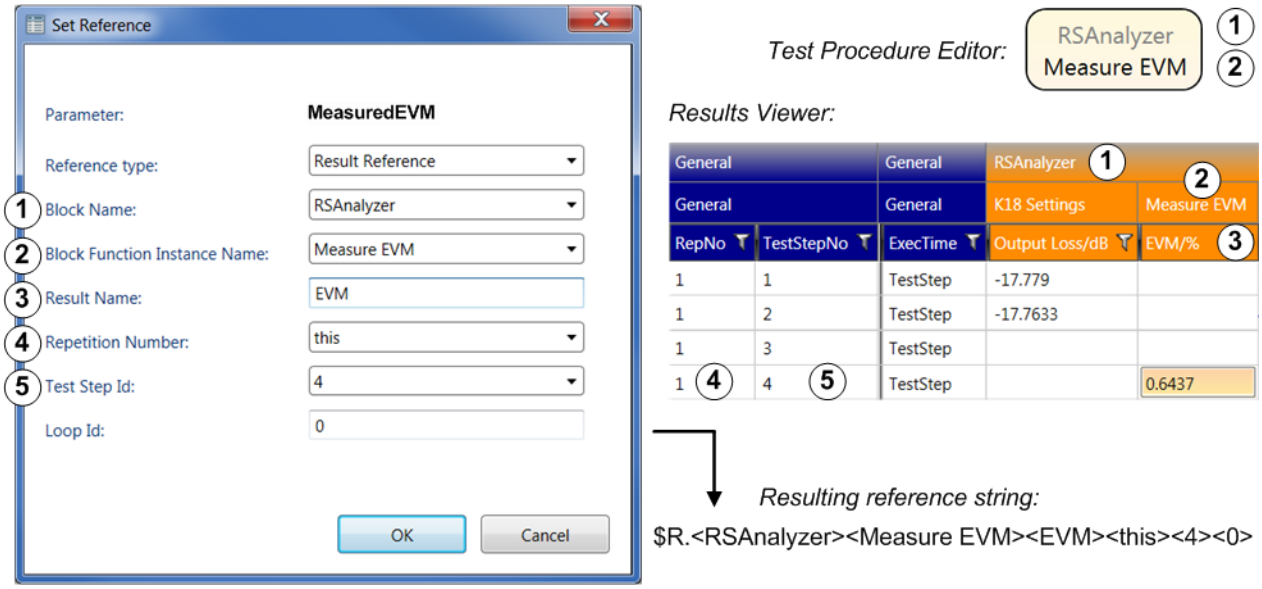


Figure 9-8: Set reference to a result

Set VISA string dialog

For a parameter of type VISA _Resource (expecting a VISA resource string as value), an edit dialog is provided as shown in the following figure. The dialog entries are assembled to a VISA resource string with correct syntax.

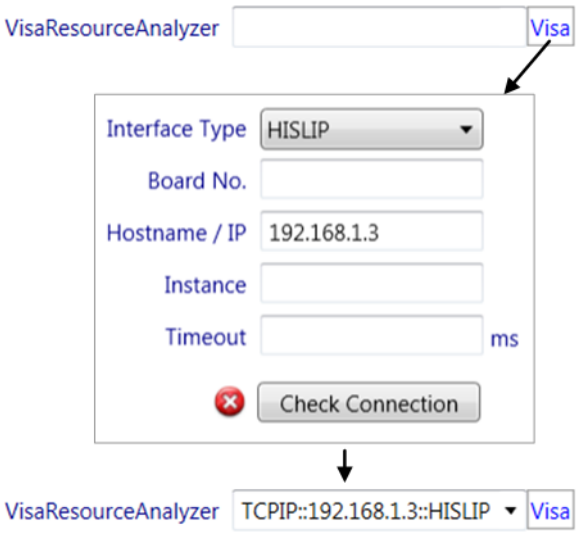


Figure 9-9: Set VISA string dialog

9.1.3 Test Execution

Test Execution consists of several sections, mainly the progress bar, the "Log Viewer" and the "Test Procedure Editor".

9.1.3.1 Progress Bar

The progress bar shows how far the test has proceeded (progress in %, current sequence and repetition). It can display a current measurement result and offers several action buttons:

- "Continue": Resumes the test execution after a halt or break.
- "Step": Executes one block function, then halts the test execution.
- "Break": Halts the test execution.
- "Abort": Stops the test execution at the end of the current test step.
- "Kill": Stops the test execution immediately.

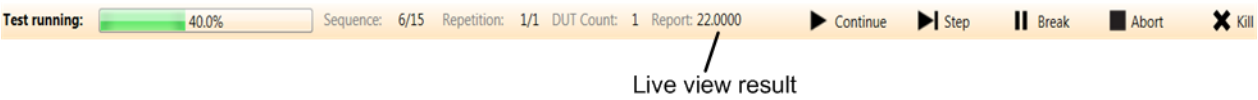


Figure 9-10: Progress bar

The result parameter whose current values are displayed in the progress bar is set in the "LiveViewReport ID" field in the "TPR Options" of the "Test Plan Editor".

9.1.3.2 Log Viewer

The "Log Viewer" protocols the events occurring during operation of QuickStep, particularly after starting the test execution. The log level defined in the "TPR Options" of the "Test Plan Editor" is taken into account. Events are connected with messages provided by QuickStep functions or components. Each event is displayed in one row with its "Timestamp", Repetition Number, Seqence ID, Source and the "Message" content.

The order of messages shown in the Log Viewer can differ in rare cases from the real execution order due to the asynchronous nature of the log messages. Refer to the Execution Protocol in the results folder to see all messages in the correct order.

Table 9-2: Toolbar

Item	Effect
"Clear"	Click this field to remove all events from the Log Viewer.
"Autoscroll"	Click this field to disable the automatic scroll down of the list of events when new events arrive. With "Autoscroll" activated, the last event is always visible.

Actions

- Click any entry in the Log Viewer and enter CTRL + F. A search window opens where you can enter a text string and search for messages containing that text string.
- Right-click a logged message. The context menu opens. Select "Copy" to copy the time stamp and message content onto the Windows clipboard.

9.1.3.3 Test Procedure Debugger

The "Test Procedure Debugger" allows to check the values of parameters during a test run. The view contains the Test Procedure Browser from the Test Procedure Editor for selecting the block function flow chart that contains the block function of interest.

The debugger works together with the progress bar during test run and with breakpoints set in the test plan table. At each halt during test execution, the "Test Procedure Debugger" highlights the block function which will be executed next (or in the test step with the next breakpoint) and shows its parameters.

Sections of the "Test Procedure Debugger":

- The test procedure browser for selecting the execution phase of interest. The selected phase determines which block functions are displayed in the main pane.
- The main area showing the block functions for the selected phase and highlighting one block function at each test execution halt. See the table below for the meaning of the highlighting colors.

In case of stepwise test execution and when reaching the next halt, the execution time between the previous and current halt for the previously highlighted block function is displayed in that block function. At the end of the test run, the execution times counted from the previous halt are displayed in the block functions. The timing information, particularly long durations, could help in identifying programming bugs for the block function.

Table 9-3: Color scheme for highlighted block functions

Color	Meaning
"Yellow"	First block function to be executed in the test step with the next breakpoint.
"Blue"	Next block function to be executed.

You can change parameter values during a halt of test execution. If you change block function parameters in the "Properties" pane of the Test Procedure Debugger, then click "Update Test Project" in the toolbar before resuming test execution.

Table 9-4: Toolbar

Item	Effect
"Remove all Breakpoints"	Click this field to remove all breakpoints which have been set in the testplan table within the Testplan Editor. The check boxes in the testplan table are unticked.

9.1.4 Testplan Editor

9.1.4.1 Toolbar

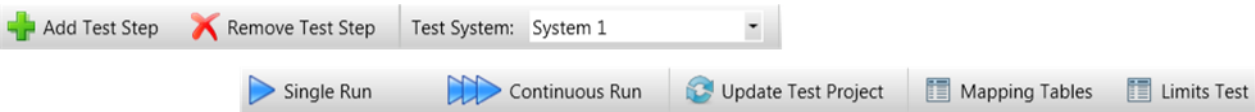


Figure 9-11: Toolbar

Table 9-5: Toolbar

Item	Effect
"Add Test Step"	Click this field to append a selectable number of test steps in the test steps table.
"Remove Test Step"	Click this field to remove that test step from the table which is currently focused (highlighted, via mouse click). If no test step has been focused, no step is removed.

Item	Effect
"Test System"	Displays the currently used system configuration and allows to select another one if available in the System Configurator. "Test System" always shows the same value as the "System Configuration" parameter in the "Mapping Table Editor". The value can be changed at both locations.
"Single Run"	Click this field to start a normal test run. The test is executed once (with all repetitions as defined in the "TPR Options").
"Continuous Run"	Click this field to start a continuous test execution. When a test run is finished (with all repetitions as defined in the "TPR Options"), the next test run is started. Each test run gets a separate results folder.
"Update Test Project"	A red "!" icon is displayed if the configuration has been changed in the "Test Procedure Editor" or "System Configurator" tab. Click this field to apply the changes to the test project.
"Mapping Tables"	Click this field to open the "Mapping Table Editor". See Mapping Table Editor below.
"Limits Test"	Click this field to open the "Set Result Limits" dialog. Here, you can import an Excel limit table and connect a result parameter with a limit parameter from the imported limit table. See Set Result Limits dialog.

Mapping Table Editor

This dialog shows the mappings of "mapping" parameters or RF paths to system configuration elements.

The "mapping" parameters are possible reference objects related to system configuration devices. After having created a mapping parameter and having mapped it to a device in a system configuration, the parameter is available as property of the device in the system configuration and its value is set there. Then, a test plan parameter value (for example) can be set as reference to the mapping parameter.

RF paths are possible name values for parameters of type RF_Path and are mapped to system configuration paths.

Parameter Mapping Table

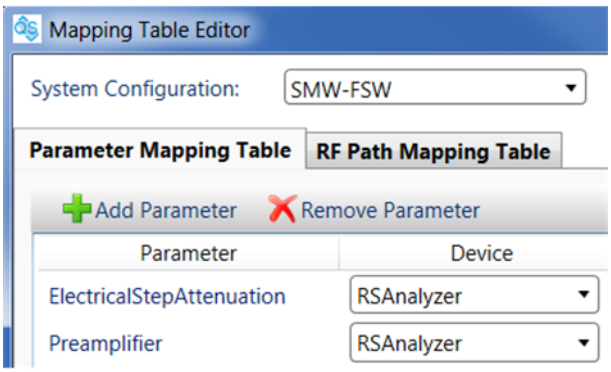


Figure 9-12: Parameter mapping

The left "Parameter" column contains parameters which appear in the mapped device of the selected system configuration. The "Device" column refers to the devices of the selected system configuration. If the value "System" is chosen in this column, the mapped parameter is listed directly in the "System Browser" of the System Configurator and is not pinned to a certain element of this system.

Actions:

- Click the "Add" button to append another mapping parameter.
- Select a parameter from the "Parameter ID" column and click the "Remove" button to remove the parameter row.

RF Path Mapping Table

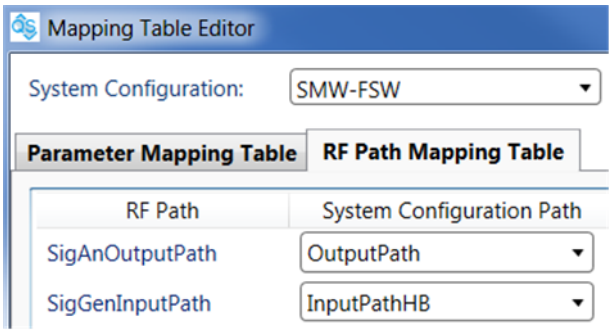


Figure 9-13: RF path mapping

The left "RF Path" column contains name values for parameters of type RF_Path (QuickStep format adapted to system configuration paths). The "System Configuration Path" column to the right lists all paths defined in the selected system configuration. After RF path mapping, during test execution, an RF_Path parameter with an RF path name as value gets the values of the mapped system configura-

tion path. If more than one system configuration paths are available, the RF path mapping can be done separately per system.

Set Result Limits dialog

Clicking "Limits Test" on the "Testplan Editor " toolbar opens the "Set Result Limits" dialog.

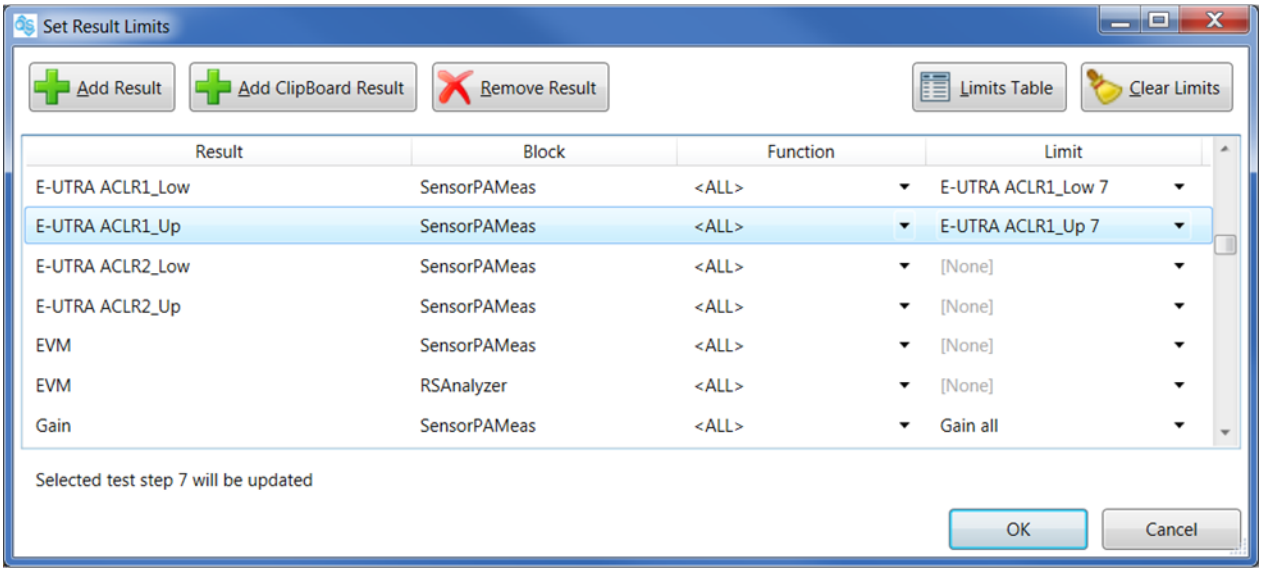


Figure 9-14: Set result limits

Action buttons:

- "Add Result": Click this button to add a result row and manually enter result name and select related block, block function and limit. The row is added in the "User-Defined Results" section.
- "Add ClipBoard Result": Click this button to add a result row with result information from the clipboard. For copying the result information onto the clipboard in appropriate format beforehand, go to the "Results Viewer", right-click the desired result and select "Copy Result Reference".
- "Remove Result": Click this button to remove the currently selected, user-defined result row. Not applicable for pre-defined results.
- "Limits Table": Click this button to have the "Limit Table" dialog opened. It shows the currently imported limit table (if available) and you can import a new one. The limit table comprises limits (names and maximum/minimum values) according to the imported source which is an Excel sheet.
- "Clear Limits": Click this button to set the values in the "Limit" column in the table below to "[None]".

The result limits table below the action buttons shows QuickStep result variables in the left column and limit names from the imported limits table in the right column. The shown result variables refer to a selected test step or sequence. The result limits table is empty if no test step or sequence has been selected or if no Excel limit table has been imported.

Action: In the "Limit" column, select and assign a limit to a result variable. The limits come from the imported limits table.

9.1.4.2 Test Project Browser and Properties

The "Test Browser" view on the left side shows substructures of the test project:

- "Test Project Parameters": Global constants and global variables which are valid for all test execution phases and test steps. The global constants are referenced by $T prefix, the global variables are referenced by $G prefix. A global variable can change its value during test execution, for example it can get its value from an output parameter of a block function via a $G reference.

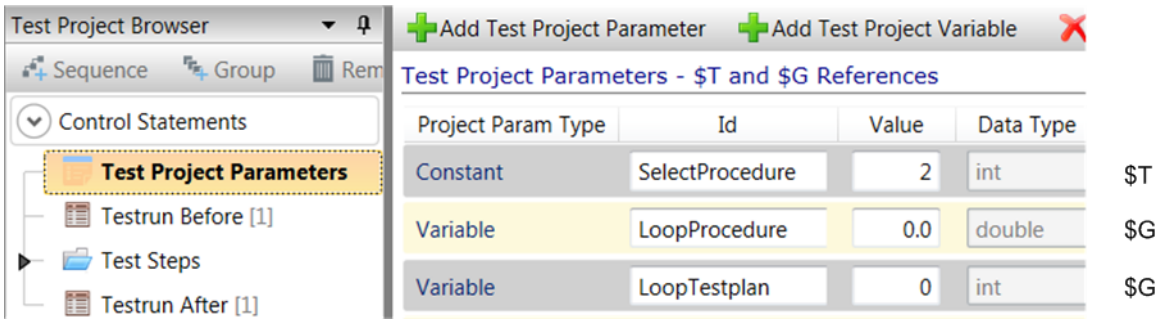


Figure 9-15: Test Project Parameters

  The tooltip shows a comprehensive list of referencing parameters and conditions.

- "Test Steps" and "Groups": Ordering structures.
- Sequences of test steps: Associated with test step tables in the main area; the number in brackets behind a sequence name indicates how many test steps are included in the sequence.
- Test execution phases (if configured) enframing the central execution of test steps; the number in brackets behind a test execution phase indicates the number of contained single procedures.

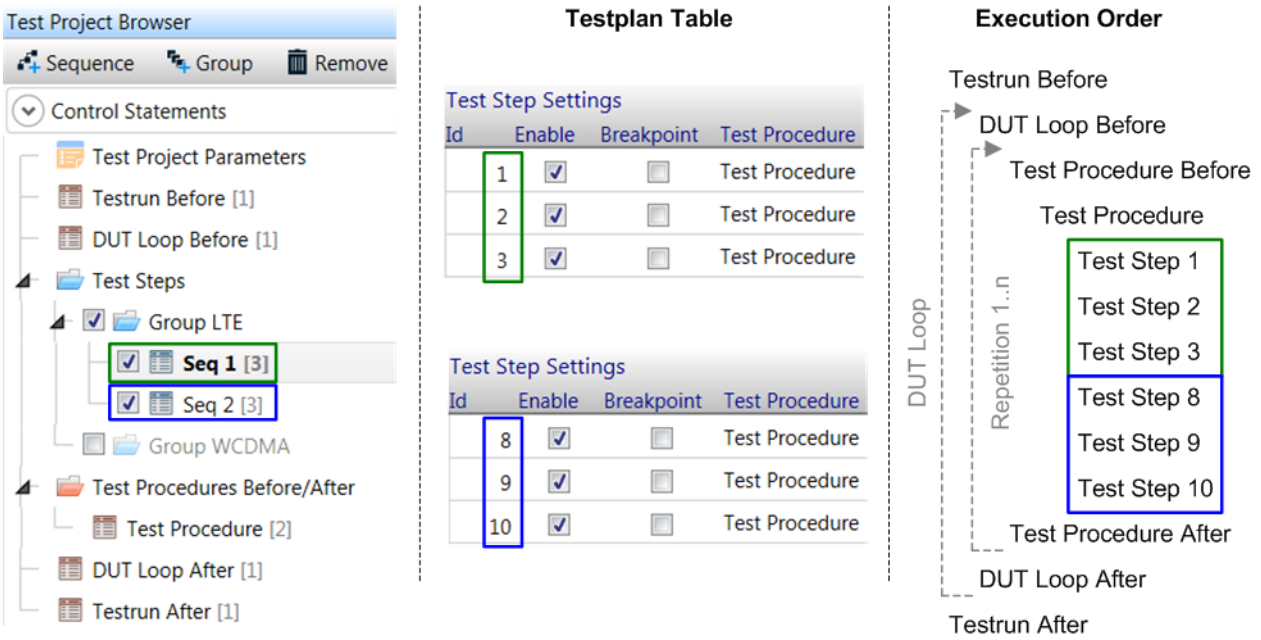


Figure 9-16: Elements of the Test Project Browser and order of execution

When clicking an element name, it is shown in bold font, its properties are displayed in the "Properties" view under the "Test Project Browser" view, and its related parameters are displayed in the middle area table. In the "Properties" view, the name, execution condition and a description of the currently focused test project element can be edited.

The groups (if available) and sequences of test steps can be activated via check boxes. All activated elements are executed during test run according to the configured conditions.

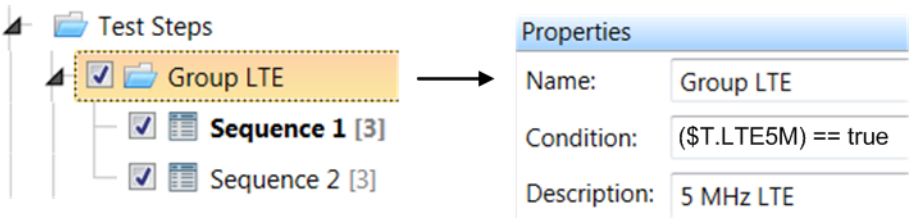


Figure 9-17: Setting properties of Test Project Browser elements

Limits for a sequence of test steps or a group can also be set.

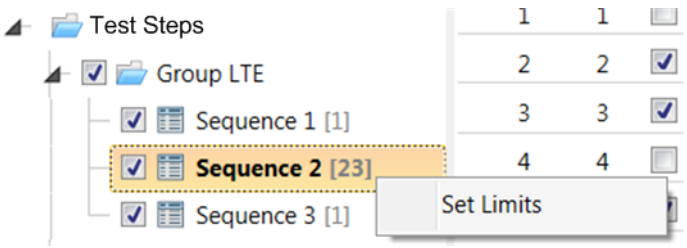


Figure 9-18: Setting limits for a complete sequence

Actions

- Activate the "Test Steps" folder via mouse click and then click "+ Group" to add a group within the folder.
- Activate a group via mouse click and then click "+ Sequence" to add a sequence under the group.
- Click a sequence name to have the test steps for that sequence displayed in the test steps table.
- Right-click a sequence to get the "Set Limits" context menu where you can set limits which are applied to all test steps of the sequence.
- Right-click a group to get the "Set Limits" context menu where you can set limits which are applied to all sequences (and all included test steps) within the group.
- Click the icon at "Control Statements" to see the control statement buttons.
  - For loops and *if* (*else*) conditions:
    Select the "Test Steps" or a "Group" folder or an already existing control element and click a loop button or an *if* condition button to add a respective control subelement (indented). If other subelements of the same level are already available, the new control element is the last one. Drag and drop the test sequences to be iterated or executed under *if* condition into the control element. Set the required parameters in the "Test Step Parameters" tab on the right side.
    You may also select a test sequence before clicking a loop or *if* button. In this case, the new control row is appended on the same level (subelement of the next higher "Group", control element or "Test Steps").
  - For "Expression":
    Add an expression subelement in the same way as an *if* subelement. The expression is applied to all test sequences under the parent element.
  - For Jumps (continue, break, abort):

Select a loop element and click a jump button to add a jump subelement (indented). Set the jump condition in the "Test Step Parameters" tab on the right side.
Jumps allow to leave a loop or immediately proceed to the next iteration.

See Control statements for more details.

Control statements

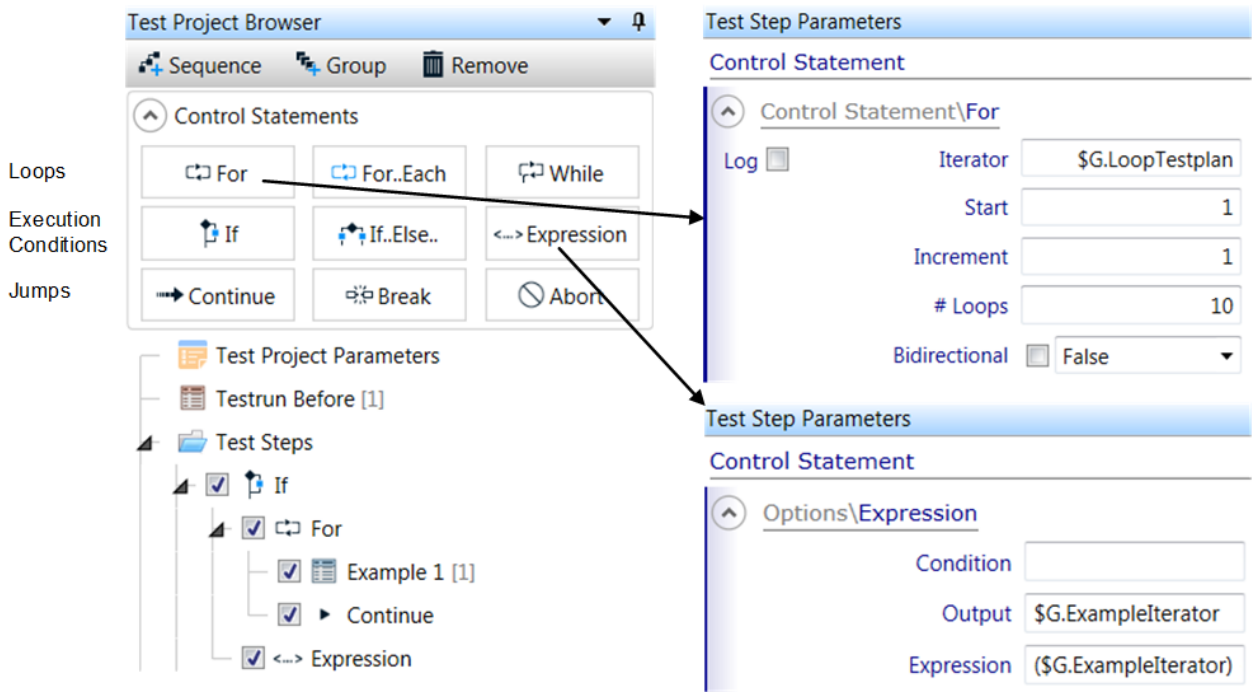


Figure 9-19: Control statements in the Test Project Browser

The control statements are configured with parametes in the "Test Step Parameters" tab on the right side.

Loops:

- "For": A test sequence under "For" is executed a number of times defined by "# Loops". The loop "Iterator" is a global variable ($G. ...) initialized with the "Start" value and getting an "Increment" at each iteration. "Bidirectional" may be useful if the loop is executed several times. The first time the loop starts at the start value and proceeds with the configured increment, the second time the loop starts with the stop value and proceeds with inverted increment.
- "For..Each": A test sequence under "For Each" is executed for a number of elements of a global array. "Iterator" defines the global array ($G. ...), "Items" the number of iterations through array elements.

- "While": A test sequence under "While" is executed with repetitions as long as the expression in "Condition" is true. The dynamic part of the expression in "Condition" is the "Iterator" which is a global variable. "Set Iterator" defines the global variable's value.

Selections:

- "If": A test sequence under "If" is executed if the expression in "Condition" is true.
- "If..Else": The "If" statement is accompanied by the "Else" statement. A test sequence under "Else" is executed if the expression in the "Condition" of the "If" element is false.
- "Expression": A "Condition" is applied to all test sequences under the parent element of the "Expression" row. If the condition is true, a value is returned via the global parameter defined in "Output". "Expression" defines the value of the global variable.

Jumps:

- "Continue": If the expression in the continue "Condition" is true, the remaining steps of the current iteration are skipped and the execution continues with the next iteration.
- "Break": If the expression in the break "Condition" is true, the loop is leaved and execution continues with the next step outside the loop.
- "Abort": If the expression in the abort "Condition" is true, the test execution is stopped.

9.1.4.3 Table Area

For Test Project Parameters - $T and $G References

Having clicked "Test Project Parameters" in the "Test Project Browser", the test project parameters and variables are displayed. These are valid for the whole test project (global parameters and variables). Test project parameters are static while the values of test project variables can change during test execution.

The list is limited to 12 entries.

Table 9-6: Table columns

Column	Meaning
"Project Param Type"	• "Constant": Static parameter with fixed value during test run. • "Variable": Dynamic parameter whose value can change during test run.
"Id"	The identifying name of the parameter for referencing to this parameter.
"Value"	• Constant: The value used during test run. • Variable: The initial value used until another value is assigned during test run.
"Data Type"	Data type of the variable or constant (default: char[]). In case of a variable, the data type is automatically adjusted with "Update Test Project" if an out parameter of a block function references to the variable (with $G): The data type of the out parameter defines the global variable's data type.

Actions:

- Click the "Add Test Project Parameter/Variable" button to add a Constant/Variable row in the table.
- Select a row and click "Remove" to remove the row from the table.
- Click the "Resolve References" button to update the references to the static parameters: New values of the parameters are updated at all referencing parameters which use these values by reference.
- Right-click into a value field to get the "Cut", "Copy", "Paste" commands.
- Right-click the colored area of a parameter's row (outside the value fields) to get the "Copy reference param" command. This command copies the complete reference string ($T.<Id> for a Constant, $G.<Id> for a Variable) onto the clipboard.
- Hover over a parameter row to get a tooltip showing where the parameter is used.

How to: See Chapter 6.5, "Using a Block Function Result as Input for Another Block Function", on page 101.

For execution phase parameters

Having clicked an execution phase (like "Testrun Before") in the "Test Project Browser", the phase parameters are displayed according to the selected blocks and block functions in the "Test Procedure Editor" for that phase. Parameters which have been kept pre-defined in the "Test Procedure Editor" (activated check box) do not appear in the table.

For a sequence of test steps

Having clicked a sequence item in the "Test Project Browser", the table displays the test steps in rows while its parameters are ordered in columns. The table header is structured in two levels. The lower level displays the parameter names, one per column, the upper levels provide a parameter grouping based on block and block function. The upper levels are of the following types:

- "Test Step Settings": Contains the general parameters independent of the used block functions. See the table below.
- "Test Procedure Parameters \ Test Step": Contains the related test procedure parameters defined for this execution phase. Their values are typically used by several block functions (usually different blocks) of the related test procedure. This parameter group avoids that a common parameter like the RF frequency which is relevant for both the generator and the analyzer has to be set separately in both blocks. The test step parameters are defined in the "Test Procedure Editor".
- "[BlockName] \ [BlockFunction]": Contains the parameters related to a block function as defined in the connected test procedure.

Table 9-7: General parameters for test steps

Item	Effect
"No"	The row number in the test steps table. This number cannot be edited.
"Id"	The unique identifying number of an individual test step in the table. Repetitions of a test step have the same Id. The Id is generated automatically and cannot be edited. If a test step is moved in the sequence, the Id stays with the step.
"Enable"	If activated for a test step, the test step is included in the test execution.
"Breakpoint"	If activated for a test step and the breakpoint mechanism has been enabled (via "TPR Options"), the test execution is paused just before this test step is executed. Resume the test execution by clicking the "Continue" button in the progress bar or execute the next block function by clicking the "Step" button.
"Test Procedure"	The selected test procedure is connected with the test step. The test procedure determines the available parameter groups and parameters, the measurements and results and the procedures to be applied.

Actions for a test steps table

- Double-click a row to activate the edit mode.
- In edit mode, hover over an input value to get tooltip information, for example a parameter description.

- Drag & drop a row to move it vertically in the table. The moved test step keeps its Id.
- Right-click a row (test step) in the table to open the context dialog which offers the following options:
  - "Sweep Value": For adding a parameter sweep. The right-clicked row is the starting test step for the parameter sweep.
  - "Set Value": For setting the value of a parameter in the right-clicked row.
  - "Single-Line Sweep": For adding a parameter sweep in a single table cell. With this option, a loop is executed for the related test step as defined with the single-line sweep settings. For each loop iteration, the complete test step actions are carried out and results are logged.
  - "Set Limits": See Set Result Limits dialog.

  For the first three options, the "Edit Test Step" dialog opens.

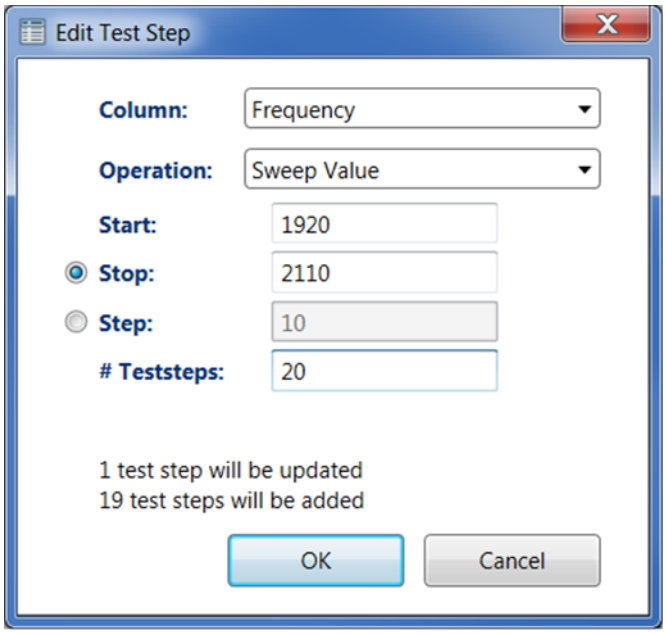


Figure 9-20: Edit Test Step dialog

Edit Test Step dialog for "Sweep Value":

- "Column": Selects the parameter to be swept.
- "Operation": Selects the functionality to be applied and determines which items are displayed beneath.
- "Start": The value of the sweeping parameter in the right-clicked row.
- "Stop": The value of the sweeping parameter in the last row of the sweep interval.

– "Step": The increment/decrement from one row to the next one within the sweep interval.
– "Count": The number of rows where sweeping of the parameter is realized. The first of these rows is the right-clicked one, the others are appended.
– "Priority" (for single-line sweep only): Determines the execution order of single-line sweeps for the case that a test step contains several single-line sweeps. Such multiple sweep loops are nested. Regarding two inline-sweeps (for example with priority values 1 and 2), the sweep with the higher priority number (2) is carried out in the inner loop.

9.1.4.4 Parameter Views on the Right Side

The different views are selected via tab on the bottom.

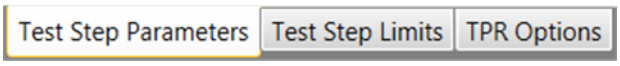


Figure 9-21: Tabs for parameter views

Test Step Parameters

In this view, the parameters of the test steps table are displayed in a vertical order. Unit information is shown if available. An additional "Description" field (not shown in the test steps table) is provided for a step-specific comment.

Hovering over a test step parameter displays tooltip information and the "R" button just at the right edge of the parameter value field. "R" resets the parameter to the default value.

Tip: The "Test Step Parameters" view is more convenient for editing parameter values than the test steps table.

Actions:

- Right-click the input field for a parameter and select "Set Reference" from the context menu to set the parameter value by reference. See "Set Reference dialog" on page 209 for details.

Test Step Limits

This view lists the result parameters which have been connected to limit parameters. The result parameters are displayed in the left column, the connected limit parameters in the right column.

Result parameters without limits are not displayed. The content of the view cannot be edited and is intended for showing information only.

Test Project (TPR) Options

Table 9-8: Test Project (TPR) options

Item	Effect
Options	
"Repetitions"	The number of test procedure repetitions including the phases "Test Procedure Before", "Test Procedure" and "Test Procedure After".
"Enable Limits"	If enabled and set to "True", the measurement results are compared with the connected limits and failures are reported. If disabled or set to "False", no limit checks are carried out for the complete test steps table.
"Continue on Limit Fail"	If enabled and set to "True", a limit failure in a test step does not stop the test execution. The failure is just reported and the test execution continues.
"Enable Progress Bar"	If enabled and set to "True", a progress bar is displayed during test execution.
"Enable Check-Block"	If enabled, the CheckBlock function is carried out for each block at the begin of a test run. Additionally, the “Init“ block function and the “Close” function are called before and after CheckBlock execution if these functions are used at any point of the test procedure. The CheckBlock function is not visible in the Test Procedure Editor. For user-defined blocks, the CheckBlock functionality has to be implemented manually via Visual Studio (for example version checks for firmware or hardware).
Debugging	
"Enable BDF Check"	If enabled and set to "True", the consistency between the block definitions for the current test procedure (collected in `*.bdf` files) and the associated block libraries (`*.dll` file) is checked at start of a test run (only relevant for user-developed blocks whose functions are implemented with Visual Studio (Express).
"Enable Breakpoints"	If activated and set to "True", the test execution halts whenever a breakpoint (set in the test steps table) is reached during test run. Resume the test execution by clicking the "Continue" button in the progress bar.

Item	Effect
"SCPI Commands"	Specifies the logging of SCPI commands: • "None": SCPI commands do not appear in the `Exec.log` file. • "Write to Exec.log": SCPI commands are logged in the `Exec.log` file. • "Append OPC/SysError": When the SCPI commands of a SCPI command string have been sent, an “OPC?” query is automatically added for SCPI write commands and logged in the `Exec.log` file. Additionally, the SYST:ERRor? query is added to every SCPI read command and the received answers are logged. • "Split and append OPC/SysError": In addition to the "Append OPC/SysError" functionality, commands which are entered in one line are split and sent separately. The SCPI command string must contain the long form of the SCPI commands to allow the splitting. Example: `SendSCPI(Command1; :Commandtree2:Command2)`
Logging	
"Logging Path"	The path and directory where the results of the test run are stored. If you leave this field empty, the results are written into the default results directory in the corresponding project directory.
"Log Level"	Defines the types of events reported in the Log Viewer and the `ExecutionProtocol.txt`. Values: • "SYSTEM/ALWAYS": Only system messages and messages with the log level ALWAYS are reported. • "ERR": Only errors are reported. • "WARN": Errors and warnings are reported. • "NORMAL": All events are reported. • "VERBOSE": Additional details like state information are also reported. • "DEBUG_PRINT": Most detailed reporting ready to be printed. When the log level is set to "NORMAL" or higher, all SCPI commands and read-backs are automatically logged in the Execution Protocol.
"LiveViewReportID"	The identifying name for the result value displayed in the progress bar during test execution. The reported result value is immediately updated with every new measurement result.
"DUT ID"	An identifying name for the device under test which is appended to the name of the result file directory. This ID can help relating result files to DUTs.

9.1.5 Results Viewer

9.1.5.1 Toolbar

Figure 9-22: Toolbar

Table 9-9: Toolbar

Item	Effect
"Browse..."	Click this field to navigate to the folder where the desired test results are stored. Having selected a test results folder, the result files are displayed in the "Result File Browser". The result folder of the last executed test is opened by default. Alternatively, a folder can be directly dropped into the "Result File Browser". The default result directory of a test project is automatically loaded when it is opened.
"Open in Explorer"	Opens a Windows Explorer which starts at the result directory for the current test. In this way, you have immediate access to the result directory and all result files. Double-click a result file to have it opened in the Windows Explorer editor.
"Export"	Click this field to store the currently loaded result table in a `*.csv` file (accessible via MS Excel or any other spreadsheet program) or `.xlsx` Excel file. The "Export Log" dialog is opened where you select the file format at "Save as type". Filter settings are also applied to the exported file.

9.1.5.2 Result File Browser

The results and logging information of a test run are distributed in several files and subdirectories.

Table 9-10: Result Files

File	Subfolder	Content
RepetitionsTimings.log	<DUT identifier>	This file lists how much time each repetition took. This information is automatically generated by QuickStep.
TestStepsResults.log		The main result file. A column in the result table contains the values for a result parameter, the table rows belong to test steps and repetitions.
TestStepsTimings.log		The main timing file. A column in the result table contains the values for a specific timing, the table rows belong to test steps and repetitions.

File	Subfolder	Content
<trace-file_name>. txt	Traces Corrected Traces	Trace result file. Each trace file stores a table where the first column contains the values of a running parameter and the second column contains the related result values (more than one result column is possible). The "Traces" folder is only available if trace result files have been recorded in a test run.
<matrix-file_name>. txt	Matrices	Matrix result file. Each matrix file stores a table where the first column and the first row contain the values of two running parameters and the inner table cells contain the result values for the row- and column-defined parameter values. The "Matrices" folder is only available if matrix result files have been recorded in a test run.
DUTLoopTimings.log	---	This file protocols the execution times logged in the phases "DUT Loop Before" and "DUT Loop After" (if available). For example, the average repetition time per DUT is protocolled.
ExecutionProtocol_000.txt		This file protocols the events (for example SCPI messages, errors) and logging information defined in the application code as occurring during test execution − depending on the applied log level. Additionally, some system information is logged automatically.
TestrunResults.log		This file protocols the results generated in the test execution phases "Testrun Before" and "Testrun After".
TestrunTimings.log		This file contains the overall time for the test execution. Additionally, other timing information is shown which was generated outside of the DUT loop.

Additionally, a copy of the test plan is included as `.tpl` file.

Actions

- Click a file in the "Result File Browser" to have its content displayed in the "Results Table".
- Drag and drop a folder from the Windows Explorer to load this folder into the "Result File Browser".
- "Expand All": Expands all subfolders shown in the "Result File Browser".
- "Collapse All": Collapses all subfolders shown in the "Result File Browser".
- Right-click from anywhere in the "Result File Browser" and select "Refresh" to have the displayed content updated.
- If the "Matrices" folder is available which contains matrix files: Select a Matrix file to load it into the result viewer. Then, right-click the matrix file and select

"Open 3D View" to have the file data visualized in a 3D diagram in a new window.

9.1.5.3 Results Table

Rows

Each row displays the measurement results for one test step.

Columns

Table 9-11: Columns for TestStepsResults and TestStepTimings

Item	Effect
"RepNo"	Indicates in which repetition of a test the measurement results have been taken.
"TestStepNo"	Running number counting the single test steps.
"TestStepId"	Identifies a test step by its Id number (the value in the "Id" column).
"LoopId"	Indicates the number of the current cycle in a single-line sweep or in other loops (with identical RepNo and TestStepId).
"ExecPhase"	Indicates which phase of the associated test procedure the row belongs to.
Each column right from the "General" columns represents one measurement parameter. The applied test procedure determines which measurement parameters are available.	

Table 9-12: Columns for ExecutionProtocol

Item	Effect
"Index"	The identifying number of a message due to a log event for the execution protocol
"Time"	The time when a log event takes place counted from the begin of test execution.
"Delta"	The time difference between the current and the previous log event.

Item	Effect
"Type"	The following values are available: • "ALWAYS"/"SYSTEM": For system messages and messages with the log level ALWAYS (defined in the TPR Options). • "NORMAL": For events which belong to the log level NORMAL and which are not related to warnings or errors (for example: execution of a block function). • "WARN": For warning messages. • "ERR": For error messages. • "EXECLOG": For all recorded messages not contained in one of the previous categories; particularly, all SCPI commands and read-backs are of this type. • "VERBOSE": For messages with detail information, for example writing to global variables, displaying output of a script. • "DEBUG_PRINT": Most detailed reports ready to be printed.
"Rep"	The repetition number.
"TestStepId"	The test step ID according to the test steps table in the Testplan Editor (the value in the "Id" column).
"LoopId"	Indicates the number of the current cycle in a loop.
"Block"	Contains the name of the block related to the reported action/event or an indicator for the processing component: "Sequencer" indicates the QuickStep application, "QuickStepEngine" indicates the runtime component.
"Log"	Contains the description of the recorded action.

Actions

- Click a column header to have the table sorted by the content of this column.
- Press the [Shift] key and click several column headers to do a multiple sorting of the selected columns in the order of your clicks (for example: first order sorting of the "RepNo", second order sorting of the test step Id).
- Click the icon in a column header to open a filter dialog for that column.
- Click in a cell in a result column to get a line chart for the result values of the column parameter. The row of the selected cell determines the "equals" value in the "Histogram and Statistics" view.
- Press the [Ctrl] key and click two (or more) cells in several columns to get line charts for the results of both (all) the related parameters in the same diagram.
- Right-click a result value or a result header and select "Copy Result Reference" to copy the result as reference string (*$R.<Block><BlockFunction><ParameterName>...*) to the clipboard. This result reference can be pasted into the value field of a test plan or test procedure parameter, for example

(only applicable if the corresponding QuickStep licenses for test procedure editing are available).

- After having loaded an execution protocol in the results table:
  - Right-click a message and select "Copy" from the context menu to copy the message onto the Windows clipboard.
  - Only for ExecutionProtocol results: Select any entry and select "Find" from the right-click menu to search for a text string.
- Move the bold vertical line with the mouse to change the area of “freezed” columns for horizontal scrolling.

9.1.5.4 Diagram

The "Diagram" displays the line chart of one or several result parameters over the test steps or over repetitions or any other running parameter. The measured data points are displayed as dots. Line charts with different units can be displayed simultaneously.

You can have line charts for one or several y-parameters:

- One y-parameter:
  In this case, one cell from a measurement parameter column has been selected in the "Results Table" view. By default, the diagram displays a line chart for the result values of this parameter over the test step number. For each repetition, there is one line chart (grouping), and the line charts are distinguished by different colors.
- Two (or more) y-parameters:
  In this case, two (or more) cells from different measurement parameter columns have been selected in the "Results Table" view. The diagram simultaneously displays the line charts for both (all) parameters. Now the line charts belonging to one parameter have one color while the line charts for the other parameters have different colors.

Y-Axis, X-Axis, Group By

The parameter (or parameters) for the y-axis are selected from the "Results Table".

For the x-axis, you select a running parameter via drop-down box which contains the parameters available in the results table.

The parameter selected for "Group By" defines which of the <x-parameter value : y-parameter value> pairs are connected in one line chart. For example,

if "RepNo" is selected for "Group By", then all <x-parameter value : y-parameter value> pairs belonging to same repetition form one line chart, and for each repetition there is one line chart.

Actions

- Right-click in the diagram to open the context menu. You can activate or deactivate "Show Legend" and "Draw Lines" or you can select "Copy Image to Clipboard".
  For large data sets, the lines and markers are disabled by default to increase the performance. Lines and markers can be enabled manually, but processing for big datasets might be slow.
- Drag one of the little white squares in the grey horizontal bar at the x-axis or in the grey vertical bar at the y-axis to change the scaling. Double-click to reset the scaling. Zooming is also possible by drawing a rectangular window while holding down the mouse button.
- Hover over a data point in the diagram to have its x value and y value displayed as tooltip.
- Click a data point to set a marker on this point. Click other data points to set more markers. The x values and y values of the marked data points are displayed. For marker 2, 3, ... the difference (delta) to the marker 1 value is also shown.
  Click again on a marked data point to remove the marker.

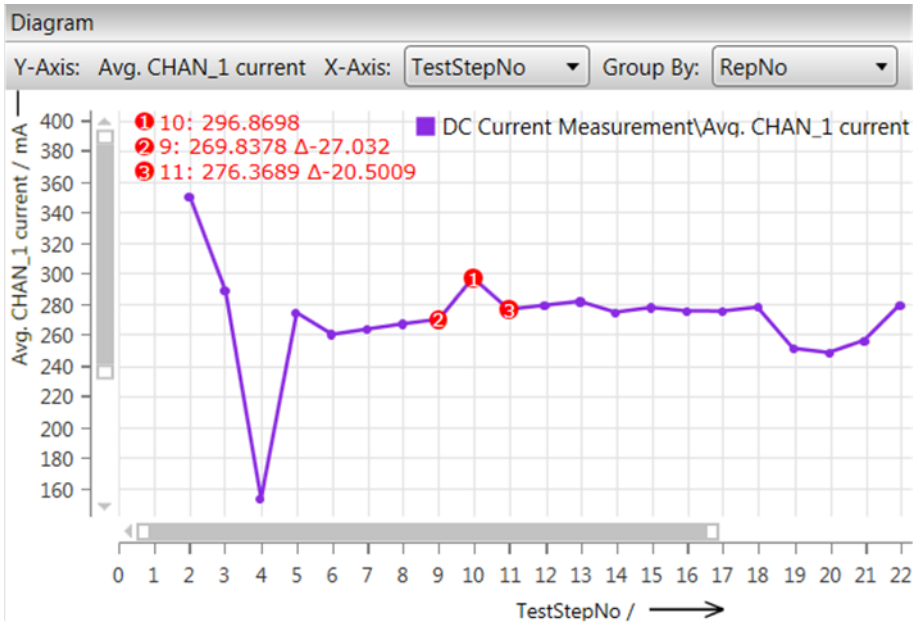


Figure 9-23: Markers in the diagram

9.1.5.5 Histogram & Statistics

"where" and "equals" define a selection of results for the histogram and the statistical values. For example "where" = "RepNo" and "equals" = 3 selects the results for the repetition number 3. The "Column" value and the "equals" value are determined by the clicked cell in the "Results Table" (the column value is the y-parameter in the "Diagram" view; only one column value is possible). When selecting "where" = "-" the distribution of the whole selected column is shown.

The histogram on the left side shows the distribution of the result values. The x-axis shows the range of values for the selected result parameter. The y-axis shows how often the result values lie in a small x-interval.

The right side lists statistical parameters for the value distribution of the result selection.

Actions

- Move the cursor over a column in the histogram to get the "x value : y value" of the column.
- Drag one of the little white squares in the grey horizontal bar above the x-axis to change the scaling. Double-click the grey bar to reset the scaling.

9.1.6 Test Procedure Editor

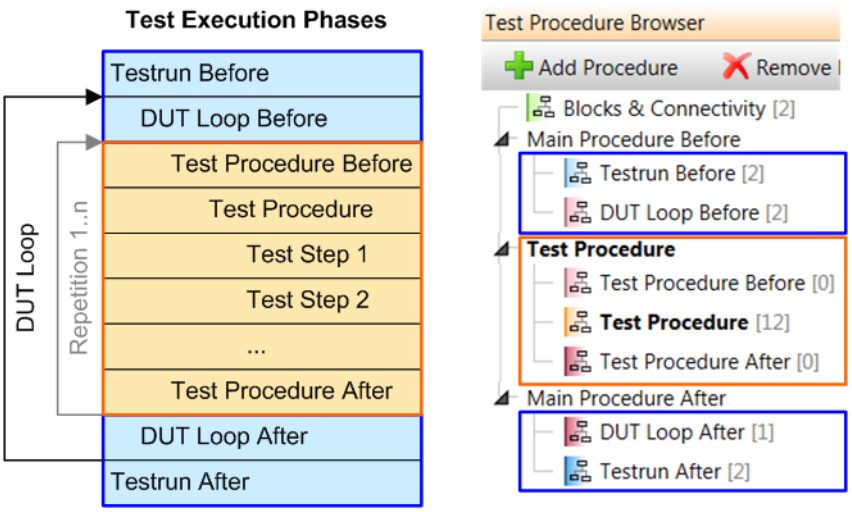


Figure 9-24: Central point of interest: Test execution phases

See Test Procedure Browser for details.

9.1.6.1 Toolbar

The toolbar provides support functions including access to assisting dialogs and tools.

Figure 9-25: Toolbar

Table 9-13: Toolbar

Item	Effect
"Reload Block Library"	Loads the block DLLs again, particularly the user-defined blocks under `C:\Users\Public\Documents\Rohde-Schwarz\QuickStep\UserBlocks\BlockLibrary` (the path might be different on your PC). Consequently, new blocks become available and modified blocks are updated.
"Forum Script"	Opens a dialog with options for creating a new Forum/Matlab script or importing an existing one. If you select "Create new script", you have to enter the name of the script and select between the public or project script folder. A script template is automatically created and you are directed to the "Edit Script Parameter" dialog to edit the script according to your needs. See Edit Script Parameter dialog for details.
"Matlab Script"	
"C++/C# Block Development"	Opens the Block Development Tool where you can generate and define a new block and its C++/C# programming structures. See Block Development Tool for details.
"SCPI Commander"	Opens a dialog where you can handle the SCPI commands for an instrument via its `.chm` help. See SCPI Commander dialog for details.
"Report Designer"	Opens the "ReportDesigner" window where you can define the appearance of QuickStep reports. The report definitions are defined in RDL, Report Definition Language. See Report Designer for details.
"Project Settings"	Configures the accessibility of tabs for different user roles. See "Project Settings dialog" on page 240.

Edit Script Parameter dialog

This dialog allows to directly add, edit and remove parameters for a script. The modifications are automatically integrated in the script.

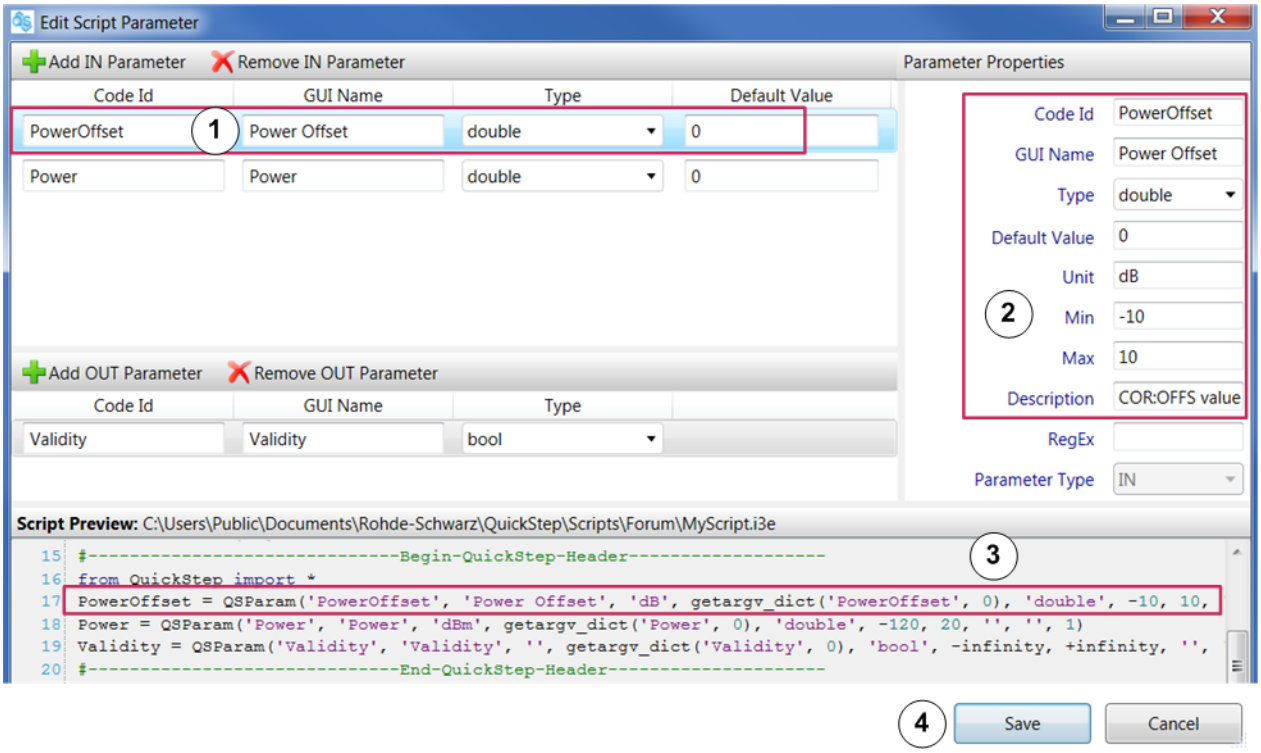


Figure 9-26: Edit Script Parameter dialog

1 = Select parameter
2 = Edit parameter properties
3 = Inspect parameter representation in script
4 = Save the script and make it available as script block function

How To: Re-Using an R&S Forum Script

SCPI Commander dialog

The SCPI commander handles the SCPI commands for an instrument via its `.chm` help. First, you import an instrument's `.chm` help. Then, you can select a command and inspect its description. You can send the command to the instrument (if a VISA connection is established) and have an automatic check whether the instrument has accepted the command without errors.

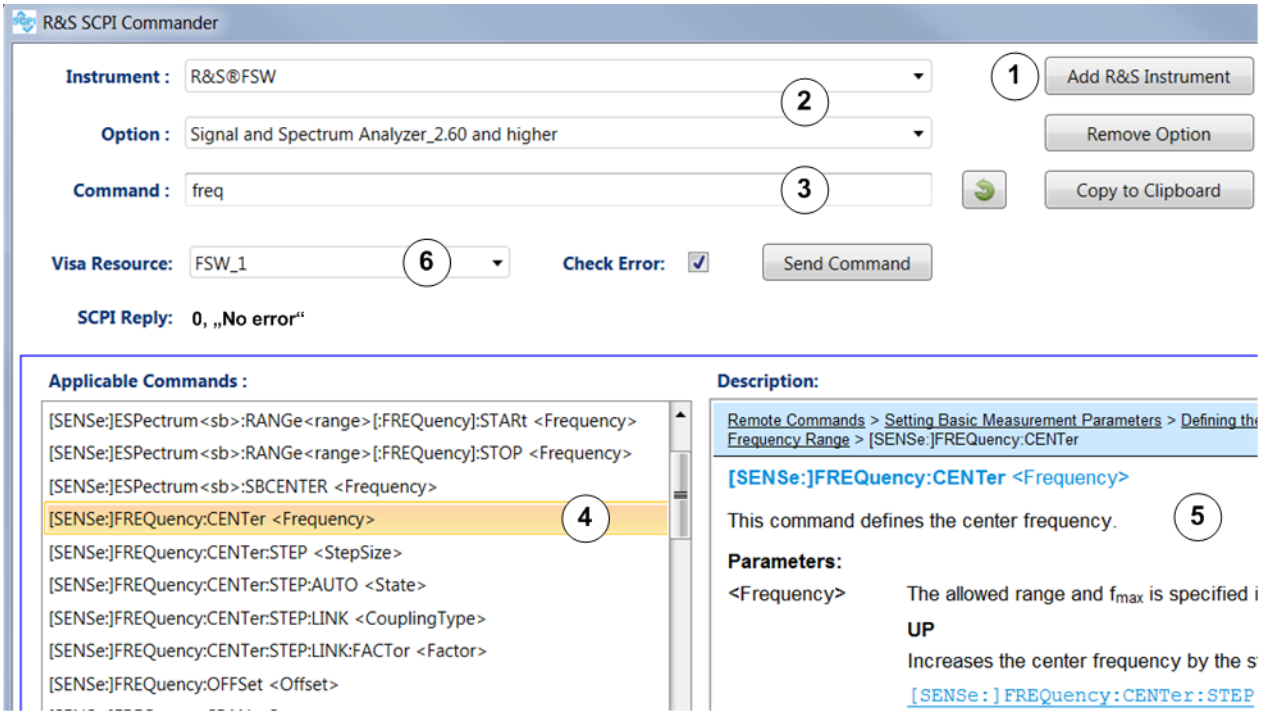


Figure 9-27: SCPI Commander

1 = Import remote command library from .chm help
2 = Select device and option to obtain the related SCPI commands (selections available after "Add R&S Instrument")
3 = Enter search string (part of a command) / edit command
4 = Select SCPI command
5 = Inspect selected SCPI command
6 = Select a VISA connection

Buttons and actions

- Click "Add R&S Instrument" to import the SCPI commands of an R&S instrument from its `.chm` help file on the file system.
- Click "Remove R&S Instrument" / "Remove Option" to remove the currently selected instrument / option from the SCPI Commander. The SCPI commands for that instrument / option are no longer available in the SCPI Commander.
- Click "Copy to Clipboard" to copy the command currently displayed.
- Click to get back from a command in the "Command" field to the previous search string.
- Click "Send Command" to send the selected SCPI command to the instrument connected by the selected Visa Resource. If the "Check Error" check box has been ticked, the SCPI commander checks the reaction to the send action and reports errors in the "SCPI Reply" field.

- Enter a search string (a part of a SCPI command) in the "Command" field to get related commands in the "Applicable Commands" field.
- Click a command in the "Applicable Commands" field to get its description in the "Description" field.
  Double-click a command in the "Applicable Commands" field to have it displayed in the "Command" field.

Report Designer

The ReportDesigner creates and edits user-defined report definitions in `.rdlx` format and style definitions in `.rdly-styles` format. These definitions can be used with "RS_Report" block functions to customize reports.

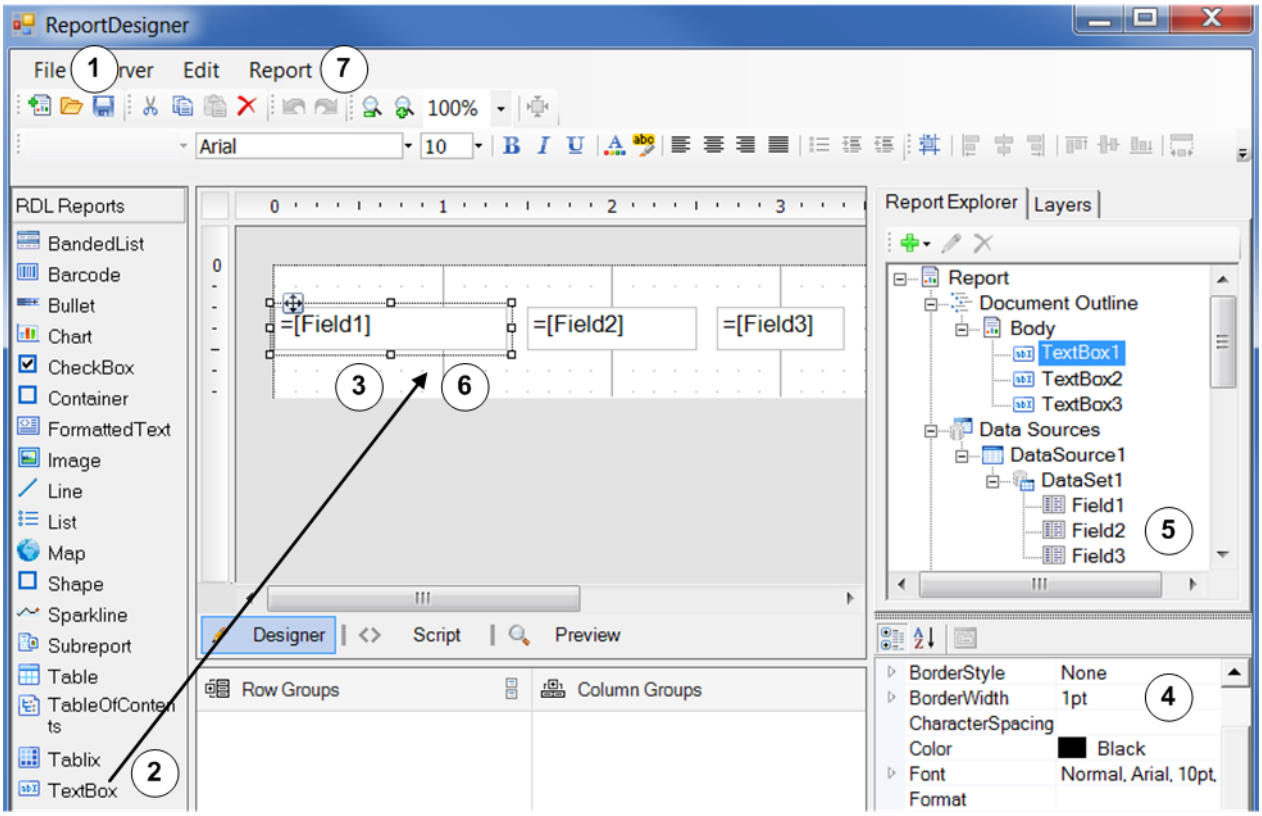


Figure 9-28: ReportDesigner

1 = Open an .rdlx report definition, save a report definition
2 = Drag a TextBox (or another element) into the report area
3 = Arrange the report elements
4 = Edit the properties of a selected report element
5 = Define fields to be filled with QuickStep data
6 = Assign a field to a TextBox
7 = Open the Stylesheet Editor for defining styles in an .rdly-styles file

Concepts: Customizing Reports

How To: Chapter 6.10, "Creating a Report Definition", on page 109, Chapter 6.11, "Creating or Modifying a Style Sheet for Reports", on page 112

For details, see the ActiveReports documentation which is included in the QuickStep documentation folder accessible via the Windows "Start" menu.

Project Settings dialog

This dialog allows to configure which parts of the QuickStep GUI are visible for a user with a typical role ("user category").

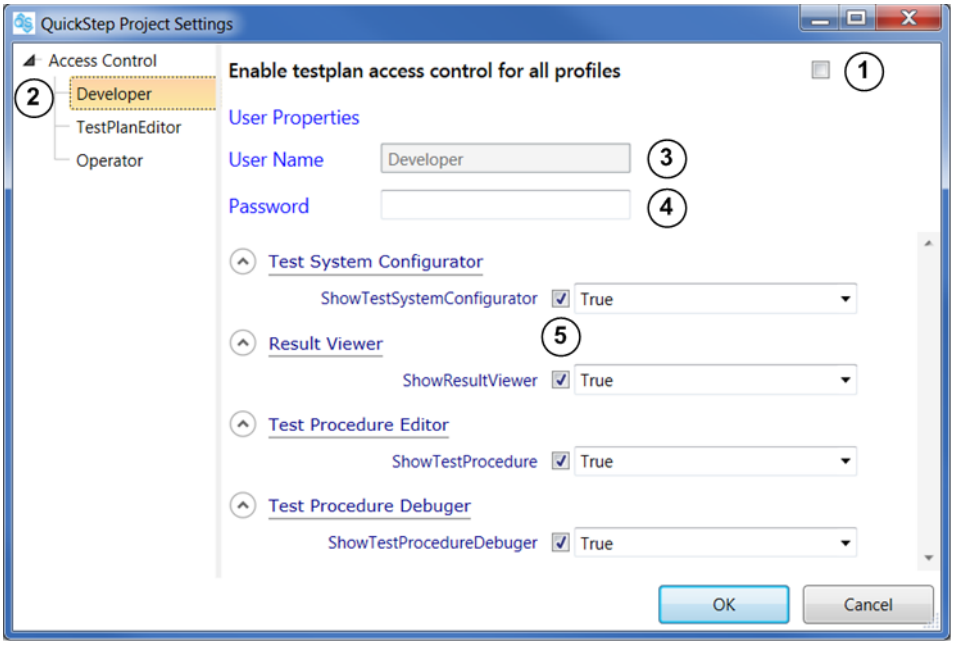


Figure 9-29: Project Settings dialog

1 = Activate access control
2 = Select user category
3 = User Name, read-only, defined by the user category
4 = Enter a password to control access for the user category
5 = Untick the checkboxes for the tabs which shall not be visible for the user category

Acess control gets active for a test plan after you have activated the check box at "Enable testplan access ...", saved and closed the test plan. Then, a login is required to open the test plan. You have to enter one of the user categories as "User Name" and the associated password (if specified). If the credentials are ok, the test plan is opened and the QuickStep tabs are shown as specified for the user type.

The QuickStep users are distinguished by the following categories. The descriptions are recommendations and may help you to select the views which shall be visible for the user.

- "Operator": The user opens a provided test plan, executes it and inspects the results.
- "TestPlanEditor": The user opens a provided test plan, edits the test steps or testplan parameters or limits and executes the test plan. Some users of this category may connect a system configuration to the test. Usually, the underlieing test procedure is taken as provided.
- "Developer": The user additionally adjusts or creates test procedures. Advanced users may develop user-defined blocks and block functions to be used in the Test Procedure Editor.

9.1.6.2 Library

This view contains the elements available for creating or modifying a test procedure. It depends on the context which blocks are displayed:

- If "Blocks & Connectivity" has been selected in the "Test Procedure Browser" on the right side, all available blocks are shown. The blocks are grouped by categories (such as User Blocks, Analyzer, Generator).
- If a test execution phase like "Test Procedure" has been selected, nodes for general elements and for available blocks are shown. Only those blocks appear which already exist in the "Blocks & Connectivity" area (exception: ForumScriptBlocks need not to be prepared in the "Blocks & Connectivity" area). Each block node contains all functions for the block as subordinate elements.

  General elements:

  – "BlockFunction": The template for any block (block type and function not specified yet).
  – "Fork/Join": Forks and Joins are used as pseudo-states. This element can be used to join the execution order of several blocks. During test execution the application waits until the functions of the blocks which end at the "Fork/Join" element have been carried out before proceeding to the following block.
  – "If": Control element with one in and two out paths. The evaluated value of the condition during test run determines which out path to a following block function is used.

– "Or": Control element with one or several in ports and one out path. When a block function connected to an in port has been executed during test run, the test execution proceeds to the next block function at the out path.
– "End":
If several branches end in one single "End" statement: The execution phase ends as soon as the block functions on all input connections to the "End" statement are completed.
If the test procedure branches terminate in several different "End" statements: The execution phase ends when the first "End" statement is reached. Already started block function are completed, but no new block functions are started in that execution phase. The following execution phases are started as usual.

Actions

- Click a block node to expand it and drag a subordinate function element into the main (middle) area. The block is added in the main area and the dragged function is automatically selected. You can reselect block type and function, subsequently.
- Drag the "BlockFunction" element into the middle area. Then select the type of block and the block function.
- "Expand All", "Close All": Expand all block nodes in the Library to have all block functions displayed, or close all nodes to have a compact listing of the block nodes.

9.1.6.3 Test Procedure Browser

This view shows the "Blocks & Connectivity" element and the phases of a test run.

- "Blocks & Connectivity": Defines the blocks usable in the test execution phases and the inter-block connections.
- "Main Procedure Before", "Main Procedure After": Contains those test execution phases which enframe the test procedure phases.
- "<Test Procedure Name>": Contains the test procedure phases, with "TestProcedure" as the main one.

The number in the [bracket] behind a test execution phase indicates the number of block functions used in that phase.

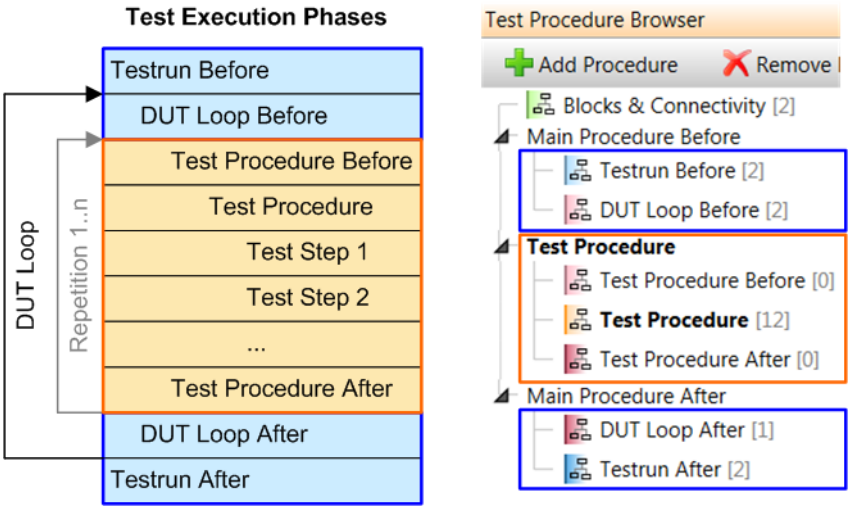


Figure 9-30: Representation of test execution phases in the Test Procedure Editor

Several test procedures (with accompanying "TestProcedure Before" and "Test-Procedure After" phases) can be defined and then used in the same test plan, see "Several test procedures used in one test plan" on page 38 for a graphical representation). In this case, the same "Main Procedure ..." actions are applied for all test procedures.

Note: If a test project is saved, the procedure configuration from the "Test Procedure Editor" is also saved.

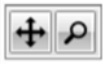
Actions

- Click a phase under a "<Test Procedure Name>" node or under the "Main Procedure ..." node to see in the main area which block functions are used in that phase.
- Click a "<Test Procedure Name>" node to have its block representation for the "Test Procedure" phase displayed in the main area.
- "Add" button: Click this button to create a new test procedure. A new and empty test procedure is appended at the end of the procedures list.
- "Remove" button: Click the button to remove the currently active test procedure (bold font). "Main Procedure ..." and "Blocks & Connectivity" cannot be removed.
- Press the [F2] key to rename a procedure.
- Click "Hide Empty" to hide all empty phases ([0] elements) of the test procedures.

9.1.6.4 Main Area

Different content is shown for "Blocks & Connectivity" and a test execution phase like "TestProcedure" or "Testrun Before" selected in the Test Procedure Browser.

"Auto fit" and "Magnifier lense":



"Auto fit": Centralizes the block representation and rescales it according to the size of the area available for the block representation.

"Magnifier lense": Displays a frame around the block representation, a zoom bar and a multiply button. When selecting a zoom factor via zoom bar, the frame is rescaled. The frame shows the edges of the representation after the actual zooming which is carried out with the multiply button.

Blocks & Connectivity

Only the block present in this area are available for the test execution phases (exception: ForumScriptBlocks). Additionally, the connections between blocks are defined here for inter-block communication.

Testrun Before, ..., Test Procedure, ..., DUT Loop After, ...

These tabs contain the blocks with their selected block functions to be executed in the different test execution phases and the dependencies between the block functions shown as arrows.

Actions

For a test execution phase tab (like "TestProcedure"):

- Position the cursor in the main area and turn the mouse wheel to zoom in and out.
- Double-click a block. Then you can select the block type and the block function via a drop-down menu. A search function is integrated in the drop-down menu: Enter the essential part of the block function to reduce the list of selectable items accordingly. Auto-complete is also supported.

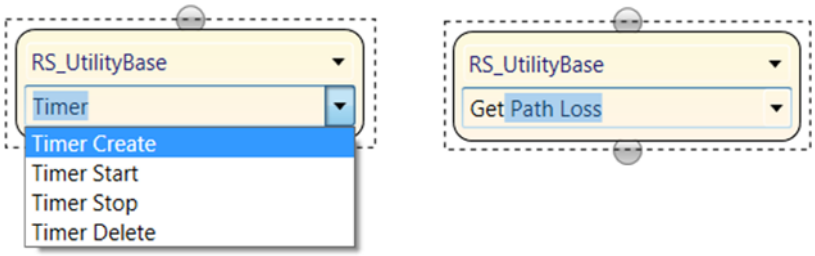


Figure 9-31: Search box for block functions

Note: Block functions where the block DLL is missing are highlighted.

- Click a block, press down the mouse key on the block and move the mouse in order to move and position the block.
- Cut, copy, paste, remove blocks with key combinations ([Ctrl+x], [Ctrl+c], [Ctrl +v], [Del]) or with the context menu via right-clicking a block or the test procedure area.
- For establishing an antecessor-successor dependency between two blocks:
  - Click the antecessor block.
  - Position the cursor over a connector element on the block edge.
  - Press down the mouse key and move the cursor to a connector element on the edge of the successor block.

  An arrow line between the two blocks is drawn.
- If a condition has been defined for a block: Hover the mouse over the name of the block to see the expression for the condition.

For block connections in the "Blocks & Connectivity" tab:

- Creating a connection between two blocks:
  - Position the cursor over the relevant port of the first block, press and hold down the left mouse key.
  - While holding down the mouse key move the cursor to a port of the second block. Then release the mouse key.

  An arrow line from the out port to the in port is drawn. The arrow has to start at the block that is sending a request. The end has to be at the block that handles the request.
- Click an arrow line and provide a name in the "Properties" area. The name is shown close to the arrow line.

9.1.6.5 Properties

This view displays one of two parameter collections: Test execution phase parameters and block function parameters.

Test execution phase properties

If you click a test execution phase in the "Test Procedure Browser", the procedure parameters for that phase are displayed. These parameters are not related to blocks but are common for the phase and the procedure. The parameters of the "Test Procedure" phase are also displayed in the "Test Plan Editor" under the "Test Step Parameters" header. If a parameter is referenced by other parameters, the background color of that parameter row is yellow. If it is not referenced by any parameter, the background color is grey.

Table 9-14: Test execution phase properties

Item	Meaning
"Def."	Check box to set the parameter as pre-defined and equal for all test steps. If checked, the parameter is not displayed in the "Test Plan Editor".
"Id"	The unique identifying name of the parameter used by the software and for referencing to this parameter.
"Value"	The value used for processing.

Block function properties

If you click a block in the main (middle) view, the properties of the selected block function are displayed. The "Properties" view contains not only general settings and the block function parameters but also a brief description of the block function and some details.

- Section with headline "[block name]\[original function name]":

  Possible configurations:

  – You can enable or disable the block function via check box. A disabled block function is skipped during test run.

  – You can modify the name of the selected block function.

  – You can add an execution condition which must be true in order to have the block function executed.

  – For ForumScriptBlocks: The "Open in Forum" and "Edit Script Parameters" buttons are displayed (“greyed out” if R&S Forum is not installed). See the Actions section below.

- "In Parameters" section:
  - "Log" check box: If checked, the parameter and its value is included in the result log.
  - "Def." check box: If checked (default value), the parameter is not displayed in the "Test Plan Editor". With this setting, you can edit the parameter value and the parameter is "pre-defined": The parameter does not appear in the test steps table and gets the same value for all test steps.
  - Parameter values: They appear as initial values in the test steps table (if the "Def." check box is not activated).
- "Out Parameters" section: The same GUI elements are displayed as for the "In Parameters". You can use out parameters to assign their values to test project variables which may serve as input for following block functions.
- "Description" and "Details" sections: Short description of the block function and details about it.

Actions

If parameters of a block function are displayed:

- Hover over a parameter name or its value field to get tooltip information and the "R" button right at the value field. Clicking "R" resets the parameter to its default value.
- For ForumScriptBlocks (and with R&S Forum installed):
  - Click "Open in Forum" to start R&S Forum where you can edit the script.
  - Click "Edit Script Parameters" to add or modify script parameters. These parameters appear in the "Properties" area and are usable in the same way as "Properties" parameters of any other blocks.

If procedure parameters of a test execution phase are displayed:

- Click the "Add" button to append a new parameter.
- Click a parameter row and then the "Remove" button to remove that parameter.
- Click the "Resolve References" button to update the references to the parameters beneath (new values of the parameters are updated at all referencing parameters which use these values by reference).
- Hover over the ID entry for a parameter to have those parameters displayed which get their values by reference to this parameter. The block and block function where a referencing parameter is located are also shown.

9.1.7 System Configurator

9.1.7.1 Toolbar

Figure 9-32: Toolbar

Table 9-15: Toolbar

Item	Effect
"Reload Library"	Reload the library after DLLs have been updated in the background, for example by modifying code of user blocks and compiling new DLLs.
"Import System"	Click this field to load a `*.sdf` system configuration from the file system.
"Export System"	Click this field to store the currently active system configuration in a `*.sdf` file.
"VISA Instruments"	Click this field to open a dialog where VISA resource strings are collected and edited, see the description below.

VISA Instruments Dialog

This dialog lists VISA resources and their aliases, and assists in setting up VISA resource strings with correct syntax.

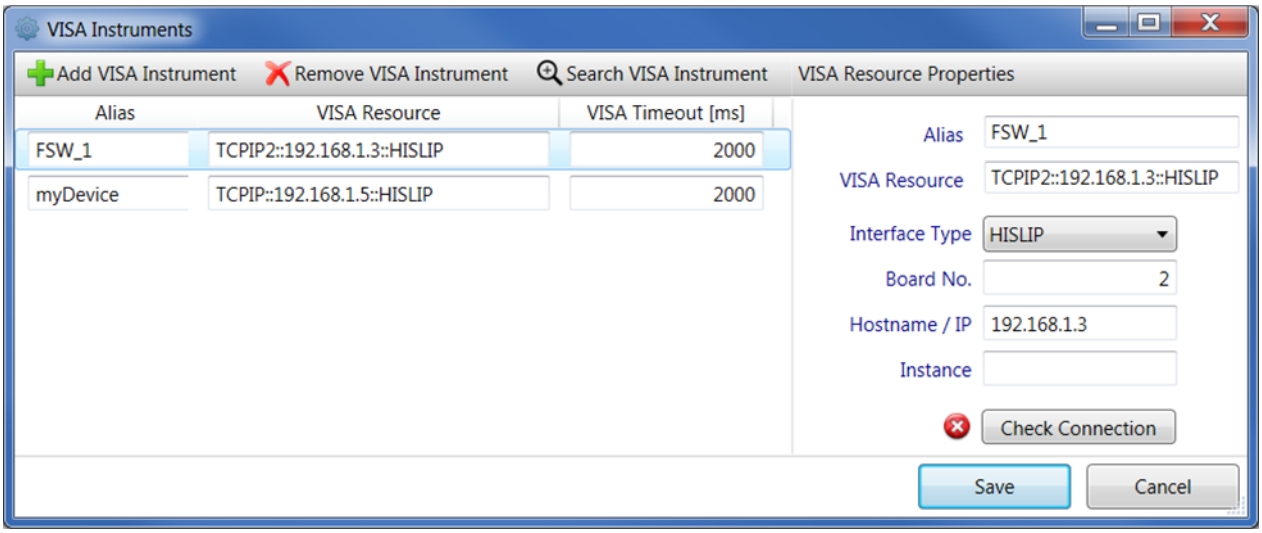


Figure 9-33: VISA Instruments dialog

Setting of a VISA resource parameter in any view is then simply done by referencing to a VISA alias.

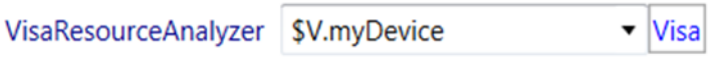


Figure 9-34: Reference to a VISA alias in a block function

Buttons in the top bar:

- "Add VISA Instrument": Appends a new row in the left dialog area with default settings.
- "Remove VISA Instrument": Removes the currently selected row in the left dialog area.
- "Search VISA Instrument": Searches for connected VISA instruments in the LAN and displays them in the "Instrument discovery" dialog. Select a detected instrument and click the "Add" button to add it to the list on the left side of the "VISA Instruments" dialog.

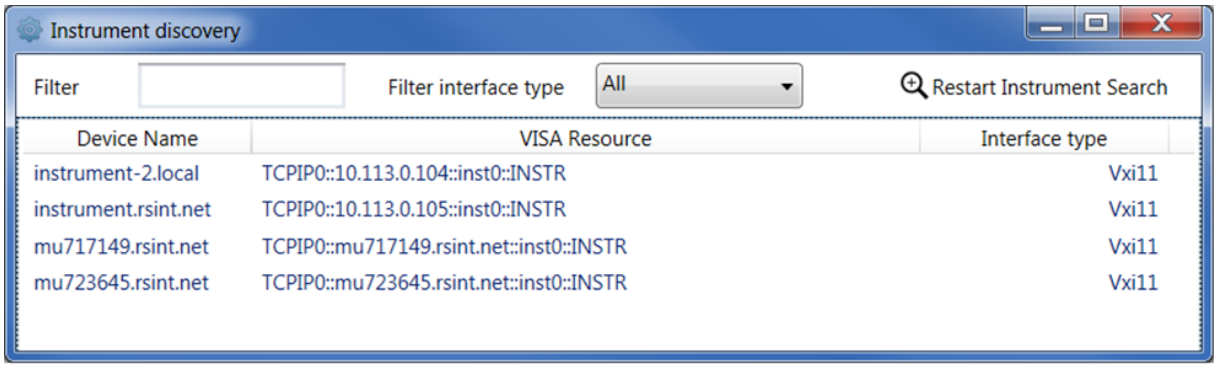


Figure 9-35: VISA instrument discovery

Left area: Shows the already configured VISA resources or a new resource with default settings. Configurations can directly be done on the left-hand side but more conveniently on the right-hand side which provides further options.

"VISA Timeout [ms]": Maximum waiting time for an answer of the instrument to a connection request or any VISA command. If the instrument does not answer in this time period, a connection failure is reported.

Right area: Displays the properties of the VISA resource selected on the left.

"VISA Resource Properties":

- "Alias": The identifying name used when referencing to the VISA resource string.

- "VISA Resource": The VISA resource string resulting from selections below (Interface Type and Hostname / IP).
- "Board No.": Only needed if more than one interface is present in the system, for example if two GPIB plug-in boards are connected to one controller.
- "Hostname / IP": The name or IP address of the test instrument to be accessed via VISA.
- "Instance": Optional parameter, typically not required. See VISA documentation for additional information.
- "Check Connection" button: When clicking this button, QuickStep attempts to establish the connection to the target instrument using the string of the "VISA Resource" parameter and sends an "*IDN?" request. The result of the attempt is reported. In case of a "Check failed" result, the "VISA Resource" parameter settings, typically the IP address, do not allow communication over VISA and testing is not possible.

9.1.7.2 Device Library

This view shows the available symbols which can be dragged into the main view to build the system configuration according to your test setup. Symbols can be part of user blocks or standalone elements.

The library includes the devices to be tested: "DUT" is taken for a standard device and has two in and two out ports. "R&S DUT" reflects an evaluation board with an amplifier to be tested and provides additional ports for power supply, envelope tracking and RFFE control.

9.1.7.3 Main (Project) View

This view displays the elements (test instruments, other components and DUT) of the test setup as blocks and the connections between the devices. The available out ports of a block are typically positioned on the right edge, the in ports on the left edge. So, position a generator block on the left side and an analyzer block on the right side.

"Auto fit" and "Magnifier lense":

"Auto fit": Centralizes the block representation and rescales it according to the size of the area available for the block representation.

"Magnifier lense": Displays a frame around the block representation, a zoom bar and a multiply button. When selecting a zoom factor via zoom bar, the frame is rescaled. The frame shows the edges of the representation after the actual zooming which is carried out with the multiply button.

Actions

- For drawing a connection line from an out port to an in port:
  - Position the cursor over the out port (orange circle), press and hold down the left mouse key.
  - While holding down the mouse key, move the cursor to the relevant in port. Then release the mouse key.
- Move a symbol to an appropriate position by dragging it while holding down the mouse key. The already existing connections with that block are automatically readjusted.
- Click a connection line that you want to move. Small handling circles are displayed at the corners of the connection line (if there are any). Drag a handling circle as you like to move the corner accordingly. These modifications are not stored, and the routing is reset on reload of the project.
- Click an instrument block or a connection line to have its properties displayed in the "Properties" view on the right side.
- Click an element in the "Main View" to activate it, then right-click in the "Main View" to get the context menu offering "Cut", "Copy" and "Remove". You can also cut, copy and remove the activated element with keys and key combinations ([Ctrl+x], [Ctrl+c], [Del]).
- After cutting or copying an element, you can paste it with the context menu "Paste" or with the standard shortcut [Ctrl+v].
- Position the cursor in the main view, press the [Ctrl] key and turn the mouse wheel to zoom in and out.
- For creating new connection points which allows to drag a line in a better position: First select the line, then hold down the [Ctrl] key and click with the mouse on the line.
  You can remove the circles in almost the same manner by holding the [Ctrl] key and clicking them.

9.1.7.4 System Browser

This view shows one or more system configurations in an abstract way with "System <n>" node(s) and subelements. If there are several system configurations, one of them is active and this one's graphical representation is shown in the main view.

Subelements of "System <n>":

- "System Parameters (Mapping)" includes the mapping parameters which are not assigned to a specific symbol in the mapping table editor (only visible if such mapping parameters are available). When activated, the mapping parameters are shown in the "Properties" area.
- The "Devices" node lists all devices and components of the test setup.
- The "Connections" node lists all created connections between out ports and in ports.
- The "Paths" node lists paths – if available –, each one consisting of a set of interrelated connections and attenuators forming an end-to-end pathway. Paths are used for referencing them from test plan parameters to obtain the loss for the active setup.

Actions

- Right-click a "System <n>" node to open the context menu. If there are several system configurations, you can select the active one in this way. Context menu values:
  - "Export"
  - "Add"
  - "Remove"
  - "Copy"
  - "Paste"
- Click a device or connection to have its properties displayed in the "Properties" view.
- Click "Add" or use the context menu to add a new and empty system configuration. A new system tree is added in the "System Browser".
- Click a new system configuration and then press the [F2] key in order to edit the name of the system configuration.

- If there is more than one system tree in the "System Browser" and you want to remove a system configuration: Click an element in the system configuration and then "Remove". Alternatively, use the context menu.
- For creating a path, right-click the "Paths" node to open the context menu and add a subordinate "Path <n>" node. Then copy the connections, which your path consists of, from the "Connections" node and paste it into the "Path <n>" node. Select the connections with a right-click and choose "Copy reference".

9.1.7.5 Properties

This view shows the parameters mapped to the specific symbol or connection which is currently focused in the main menu. You can edit or select the parameter values.

Typical mapping parameters for test instruments

Those parameters for test instruments (generator, analyzer, power supply) are displayed which were mapped to this symbol in the mapping table editor. Usually, these are the parameters necessary to establish a connection to the device and typically used in the Init function of the corresponding instrument block.

Parameters for lossy components and connections

The parameters for other devices (for example coupler, splitter) and connections define their attenuations.

Parameters:

- "Use": The source where the attenuation values are taken from. The source is either one of the following three parameters or "Subelements". The selection "Subelements" defines that the path attenuation is calculated as sum of the attenuations of the individual path elements.
- "Path Loss": Path loss or attenuation in dB (fixed frequency-independent attenuation). Enter loss values as negative numbers.
- "SnP File": The name of the `.snp` file (Touchstone file format) containing frequency-dependent attenuations.
- "CSV File": The name of the `*.csv` file containing frequency-dependent attenuations.

9.2 Block Development Tool

This tool creates new blocks and is also used for modifying existing blocks. In case of a new block, not only the block definition is created but also the environmental structures for programming its functionality.

9.2.1 Block Generator

Figure 9-36: Block generator

Table 9-16: Block generator elements

Item	Meaning/Effect
Common elements and elements for selected "Create New Block"	
"Block Name"	The name of the block to be created. Also the name of the folder containing the source code and block definition files.
"Block Type"	• "Instrument Block": The block is typically associated with a device. The block project to be created may use VISA functions and the block can have device parameters. • "Instrument Block with VISA": An instrument block for which basic VISA functionality is automatically generated: Init, Printidentity, Close SCPI_Read, SCPI_Write, SCPI_Query, SCPI_WaitTillDone block functions. • "Software Block": The block is typically not related to a device. Device parameters are not supported. • "System Configurator Block": No code is associated with this block type, only a symbol for the System Configurator is created. • "System Configurator Block with path loss": A System Configurator Block which offers the parameters to handle path losses.
"?"	Displays information about the selectable block types.
"Programming Language"	"C++ (64 bit)", "C++ (32 bit)" (legacy system support) and "C#" are selectable for the type of source code (auto-generated and user-defined) implementing the block functionality.
"VisualStudio Version"	Defines the version of MS Visual Studio (Express) used for programming the block functionality.

Item	Meaning/Effect
"Destination Path"	The directory where the new block is stored. The path ends with the specified Block Name. It is recommended to use the default path.
"..." button	Clicking "..." opens a Windows Explorer and you can select the path where the new block is stored.
"Create New Block" button	Click this button to create a new block (Visual Studio project) according to the current settings on the left.
	Selects the type of "Create ..." or "Copy ..." button and adjusts the available fields on the left: • "Create New Block" (default) • "Copy User Block": If selected, the "Source Path" field is displayed (see below). • "Copy Template Block": If selected, the "Block Template" and the "Source Path" fields are displayed (see below).
Elements for selected "Copy User Block" or "Copy Template Block"	
"Copy User Block" button	Click this button to create a new block (Visual Studio project) by copy of the existing one selected in "Source Path".
"Copy Template Block" button	Click this button to create a new block (Visual Studio project) by copy of the selected template block (see below).
"Block Template"	Available (replaces "Block Type") if "Copy Template Block" has been selected on the right side. Specifies the type of dialog block to be created by copy of a template block. Options: • "B_RS_NonBlockingGuiCsWinForm": Windows Forms dialog block based on C# and with own dialog thread (hence, other block activities are not blocked while the dialog is running). • "B_RS_NonBlockingGuiCsWpf": WPF dialog block based on C# and with own dialog thread • "B_RS_BlockingGuiCsWinForm": Windows Forms dialog block based on C# and with one shared block thread (hence, other block activities are blocked while the dialog is running) • "B_RS_BlockingGuiCsWpf": WPF dialog block based on C# and with one shared block thread See User-Defined GUI for further information about dialog blocks.
"Source Path"	Specifies the path to the block or block template to be copied. The path ends with the block folder name.

Actions

- Enter a block name, select the programming language and Visual Studio version used for implementing the block functionality. Then click "Create New Block" to create a new block in the default "Destination Path". It is automatically loaded in the Block Definition File Editor.

- Select "Copy User Block" via icon on the right side. At "Source Path", select the path to the desired existing block (ending with the block folder name) and then click the "Copy User Block" button to create a new block as copy of the selected source block.

9.2.2 Block Definition File Editor

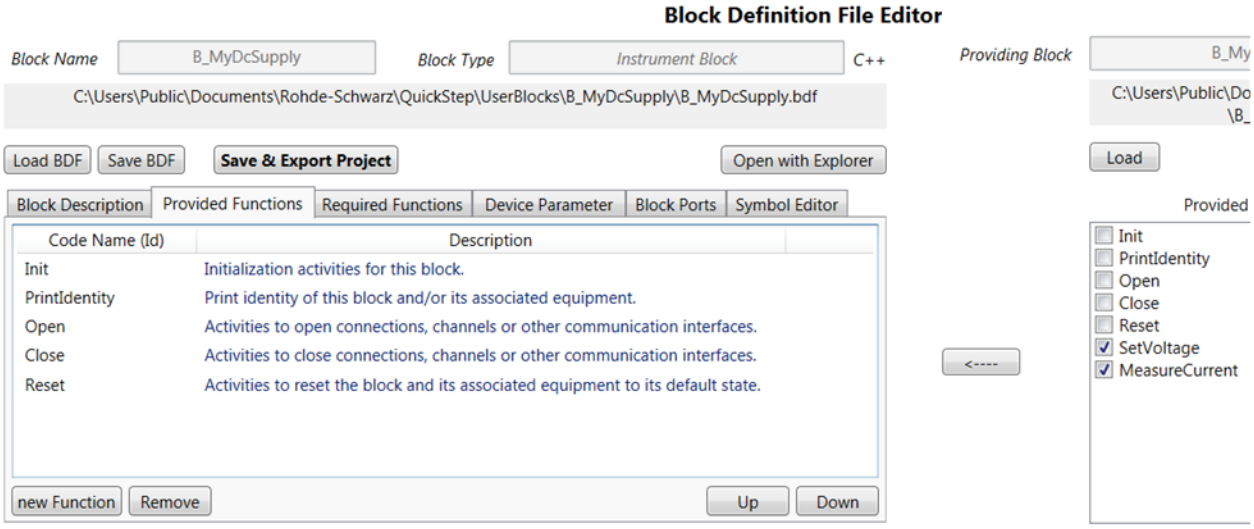


Figure 9-37: Block definition file editor

The left side displays the block to be defined or modified and its properties as well as the means for editing the properties. The right side is only used as source for copying functions from another block to the block on the left (typically as "Required Function"). Therefore, a block can be loaded on the right side. Only its available functions are displayed. These functions can be selected and copied to the left side.

Left side

Table 9-17: Elements in the upper area

Item	Meaning/Effect
"Block Name"	The name of the block which is currently loaded.
"Block Type"	Depends on the intended usage of the block. See the table above.
Path field	This editable field contains the directory of the currently loaded block.
"Load BDF" button	Opens a Windows Explorer for selecting a `*.bdf` block definition from the file system. The selected block definition is loaded into the current Block Definition Tool window. The block's name, type and directory are displayed in the upper area, its properties in the lower area.

Item	Meaning/Effect
"Save BDF" button	Only saves the block definition file with the current "Block Name", type, path, functions and parameters. Typically, "Save & Export Project" should be used during block development.
"Save & 'Export Project" button	Saves the block definition, exports function bodies of the provided functions to source code and adds project information. Renaming of a provided function leads to a new function body. New functions are added at the end of the generated code to avoid the deletion of user code. The code created under the obsolete function name is not deleted automatically. A warning is created to do this manually.
"Open with Explorer" button	Opens a Windows Explorer which starts at the block directory for the current block. In this way, you have immediate access to all available project files, for example to start Visual Studio with the project files.

Table 9-18: Tabs in the middle area

Tab	Content
"Block Description"	Input fields for brief and detailed textual description of the block's meaning and functionality as well as the block category.
"Provided Functions"	The functions defined in the block.
"Required Functions"	Functions from other blocks which can be called by the block.
"Device Parameters"	Control and configuration parameters for an instrument block (communicating with a test instrument). Device parameters can be used to store certain properties in the block and to avoid unnecessary execution of code if parameters of an instrument did not change. For example, the transmission of SCPI commands to the device can be avoided if the last sent parameter values have not been changed.
"Block Ports"	The in and out ports for connecting the block with other blocks.
"Symbol Editor"	An editor for designing the symbol which represents the block on the GUI (optional).

Table 9-19: Buttons in the bottom area

Button	Function
"New Function" /"New <Element>"	Creates a new function or a new element (the element depends on the currently selected tab).
"Remove"	Removes every activated (highlighted) row or entry in the current tab.
"Up"	Used to modify the order of items. This function also affects the order of items shown in the GUI.
"Down"	Used to modify the order of items. This function also affects the order of items shown in the GUI.

Right side

Table 9-20: Elements in the upper area

Item	Meaning/Effect
"Providing Block"	The name of the block which shall provide block functions to copy.
Path field	The directory where the providing block comes from.
"Load" button	Opens a Windows Explorer for selecting a block from the file system. After loading, the block's provided functions are displayed.

9.2.3 Tabs within the Block Definition File Editor

9.2.3.1 Block Description

This tab provides the following input fields:

- "Category": Selects a grouping node or container under which the block is arranged. The grouping node is visible in the Test Procedure Editor's Library. A new category can be defined by entering a new name.
- "Brief block description": Short information about the block.
- "Detailed block description": More information about the block.

When the block is available in QuickStep (after code implementation, compilation and project built), the contents of "Brief block description" and "Detailed block description" are shown as tooltip in the block library of the Test Procedure Editor.

9.2.3.2 Provided Functions

This tab displays the functions defined in the block itself.

Actions

- Click the "new Function" button to create a new block function displayed as "newFunction". Double-click "newFunction" to configure the new function (name, parameters, ...) with the "Function Editor", see Function Editor.
- For copying a function from a "Providing Block" already loaded on the right side: Activate the desired function on the right side via check box, then click the "<---" button.

The function appears in the view on the left side. The new function is a copy of the interface definition of the originating function (no user code is copied). It is executed as any other provided function.

Function Editor

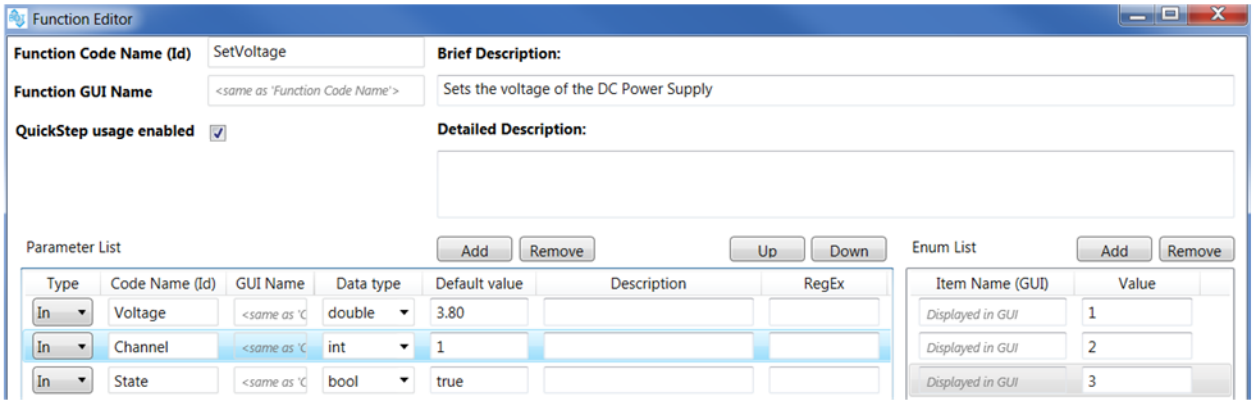


Figure 9-38: Function editor

Upper area

Table 9-21: Upper area elements

Item	Meaning
"Function Code Name (Id)"	The name of the function used in the programming code.
"Function GUI Name"	The name of the function used on the GUI.
"Providing Function (Id)" (only for "Required Functions")	The original name of the providing function of another block. Usually this name is the same as "Function Code Name (Id)". But when renaming "Function Code Name (Id)", the original name must be preserved here.
"QuickStep usage enabled"	If the check box is activated, the function will be visible in the QuickStep GUI.
"Direct DLL call enabled"	If the check box is activated, the function will be exported and can then be called from the resulting block DLL without using QuickStep.
"Brief Description"	Informative text which is shown first of all in the "Description" column in the "Provided Functions" tab. When the block is available in QuickStep (after code implementation, compilation and project built), this information will also be displayed in the "Properties" view of the "Test Procedure Editor". Therefore, the block function, present in the main view for any execution phase, has to be clicked.
"Detailed Description"	Informative text which is displayed as "Details" in the "Properties" view of the "Test Procedure Editor" when the block function is clicked in the main (middle) view.

Middle area

See the tables **Parameter list** and **Enum list** below.

In the parameter list, the properties "Code Name (Id)" and "Data type" are mandatory.

The enum list belongs to the parameter which is currently focused in the parameter list at the left.

Table 9-22: Parameter list

Property	Description
"Type"	Values: • "In": An input parameter for the related function. • "Out": A return parameter of the related function.
"Code Name (Id)"	An identifying name for the parameter used in the programming code.
"GUI Name"	The name displayed only in the GUI.
"Data type"	See Table 9-23.
"Default value"	The preset and fallback value.
"Description"	An informative text string shown as tooltip in the GUI.
"RegEx"	Regular Expression, i.e. the expected syntax of a string.
"Error message"	The warning text displayed if the parameter value does not match the expected syntax as given in "RegEx".
"Min"	The minimum value of the parameter. If you try to set a lower value, you get an indication about the expected parameter range.
"Max"	The maximum value of the parameter. If you try to set a higher value, you get an indication about the expected parameter range.

Table 9-23: Data types

Data type	Description
"bool"	Single bool value; valid values are “true” and “false”
"bool[]"	Array of bool values; length of the array has to be specified e.g. bool[5];
"byte"	Single byte value; typically only used if no other simple data type (int, bool, double, char) is suitable for the application
"byte[ ]"	Array of byte values; the length of the array has to be specified, for example byte[5]

Data type	Description
	Available types: • "char[ ]": A character string with variable length (last property in the code to avoid memory conflicts with other parameters) • "char[...]": Enables a variable argument list as parameters, i.e. multiple character arrays, e.g. for output of several strings (last property in the code to avoid memory conflicts with other parameters).
"int"	Single integer value
"int[]"	Array of integer values; the length of the array has to be specified, for example integer[5]
"double"	Single double value
"double[]"	Array of double values; the length of the array has to be specified, for example double[5]
"char[ ]"	Array of character values; the maximum length of array can be specified; if the length is not specified (dynamic length), this parameter must be the last one in the parameter list (it will automatically be moved) only one parameter with unspecified length is allowed per block function
"char[...]"	Array of character values with variable length; this parameter must be the last one in the parameter list (it will automatically be moved); only one parameter with unspecified length is allowed per block function; C++: char[…] implements the variable argument list (va_list) feature C#: char[…] is equal to char[ ]
"char[SHORTNAMELENGTH]"	Array of character values with maximum length SHORTNAMELENGTH (30)
"char[LONGNAMELENGTH]"	Array of character values with maximum length LONGNAMELENGTH (255)
"RF_Path[LONGNAMELENGTH]"	Array of character values adapted to system configuration paths; maximum length is LONGNAMELENGTH (255)
"VISA_Resource[LONGNAMELENGTH]"	Array of character values expecting a VISA resource string; maximum length is LONGNAMELENGTH (255)
SCPI_Command[LONGNAMELENGTH]	Array of character values adapted to SCPI commands; maximum length is LONGNAMELENGTH (255)

Table 9-24: Enum list

Item	Meaning
"Item Name (GUI)"	Each entry in this column is displayed as selectable item in the drop-down box for the associated parameter.
"Value"	A value connected with the item entry in the same row. While the item entry is used for selection in the drop-down menu in the GUI, the value is used by the software for actual processing.

Actions

- Click the "Add" button at the parameter list to append a new parameter row.
- Click a row in the parameter list to activate it, then click the "Remove" button to remove the activated parameter from the list.
- Click the "Add" button at the enumeration list to append a new row.
- Click a row in the parameter list to activate it, then click "Up" or "Down" to have the next or previous row activated.

9.2.3.3 Required Functions

This tab displays the functions which the block calls from other blocks.

Actions

- Adding a function from a "Providing Block" already loaded on the right side: Activate the desired function on the right side via check box, then click the "<---" button.
  The function appears on the left side. The required functions can be seen as a call of functions which are defined in other blocks. Therefore, the parameters of the imported function must not be changed as the parameter signature must match the provided function of the providing block.

9.2.3.4 Device Parameters

Device parameters store certain instrument settings and allow to avoid unnecessary transmissions of SCPI commands to the instruments instruments. They also allow to avoid any other unnecessary time-consuming activities. For detailed information about device parameters, see Chapter 7.1.5, "Device Parameters", on page 133.

Actions

- Click the "new Device Parameter" button to create a new device parameter displayed as "newParameter". Click "newParameter" to configure it with the "Device Parameter Editor", see below.

Device Parameter Editor

The figure shows an example for defining the properties of a device parameter. The properties have the same meaning as for the parameters in the Function editor, see Table 9-22.

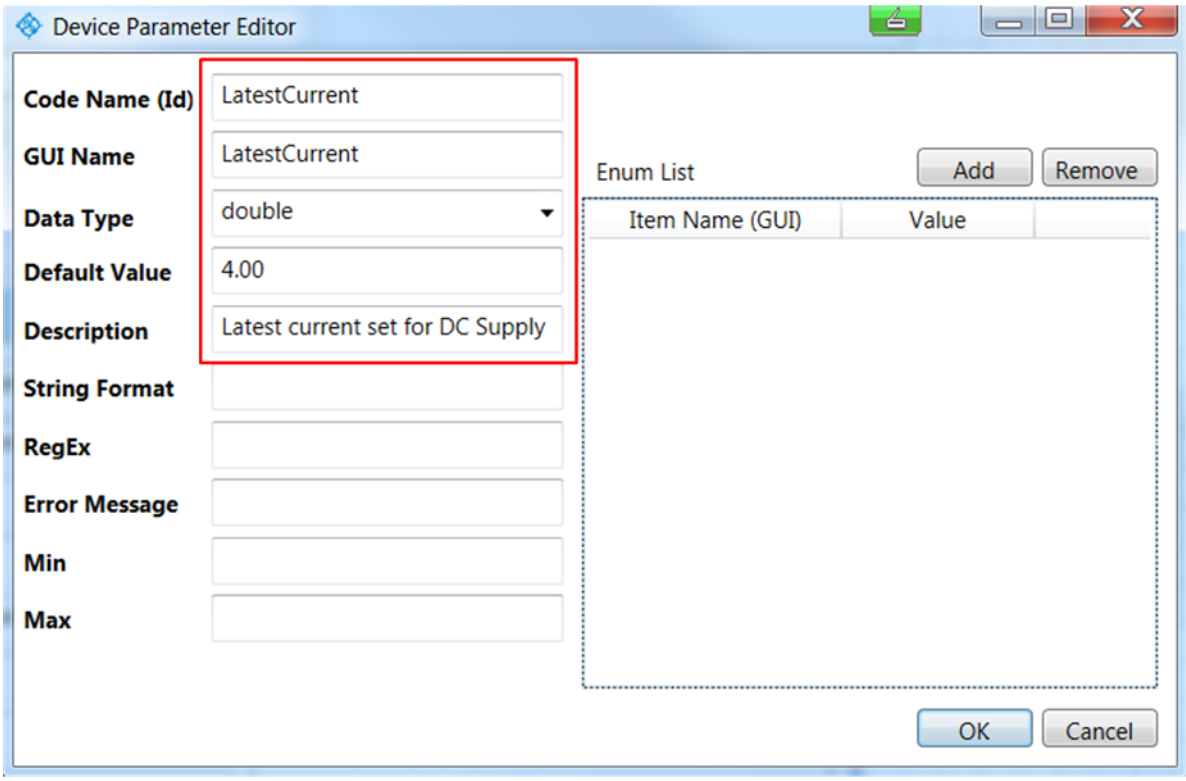


Figure 9-39: Device Parameter Editor

9.2.3.5 Symbol Editor

The "Symbol Editor" tab configures the graphical representation of a block for use in the "System Configurator". See the figure.

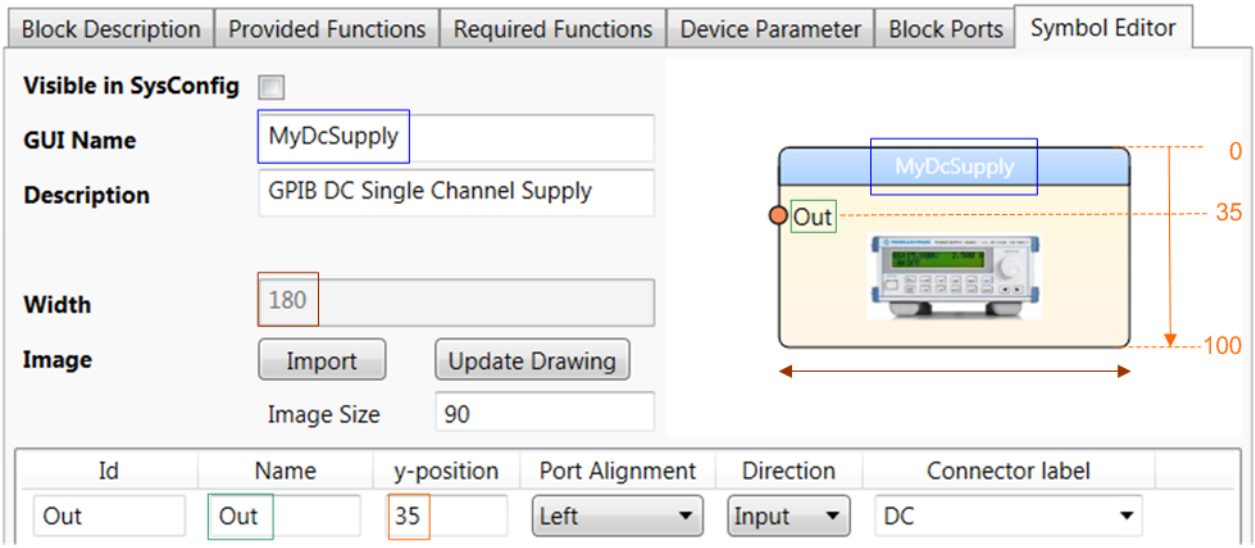


Figure 9-40: Symbol editor

Table 9-25: Configurations with the symbol editor

Item	Meaning/Effect
"Visible in SysConfig"	If activated, the block is visible in the "System Configurator".
"GUI Name"	Determines the name of the instrument block in its title bar.
"Description"	Defines the text which is displayed when hovering the mouse over the instrument block.
"Width", "Height"	Defines the size of the instrument block representation.

Item	Meaning/Effect
"Image"	• "Import" button: Opens a Windows Explorer for selecting and importing an image file. The original image is copied to the block directory and renamed to the block name. • "Update Drawing" button: Updates the graphical block representation on the right side after import of an image. • "Image Size": Defines the size of the square within the instrument block where the image is displayed.
Connector configuration area	Table columns: • "Id": The port identifier in the code. • "Name": See the figure for explanation. • "y-position": See the figure for explanation. • "Port Alignment": "Left" or "Right" specifies the side of block frame where the port symbol is displayed. • Direction: "Input" or "Output" specifies whether the block receives or sends commands/data. • "Connector label": An additional label for the connector (for documentation only). Buttons: • "New Connector": Adds a new connector row in the table. • "Remove": Removes an activated connector row from the table.

9.2.4 Info Window

This section at the bottom of the Block Development Tool window displays reports about successful or failed operational steps. If possible, required user input is reported.

9.3 Block Library

This chapter describes the functions and parameters of blocks provided with the QuickStep installation.

9.3.1 Instrument Blocks

Blocks in this chapter control single instruments.

RS...Base blocks

These blocks mainly allow to select and specify remote control (SCPI) commands which will be sent to the connected instrument during test execution. In many cases the block functions are valid for multiple types of instruments of the same class (for example different generator types). The block functions can be grouped and characterized as follows:

- Functions which execute a functionality not covered by a simple SCPI command. Examples are the Init and Close functions. Init often defines the protocol (for example HISLIP) and address where the SCPI command is sent to. Other block functions of this type execute a combination of several SCPI command at once.
- Functions providing one fixed SCPI command. If a parameter is expected for the command, the parameter value can be set.
- Functions which allow to enter a SCPI command manually. Examples are the Write SCPI and Read SCPI commands. These commands usually offer configuration parameters for logging.

9.3.1.1 RS_RfGeneratorBase

This block provides a basic set of the most common remote control commands for the control of signal generators. Standard functions like Init and Close and functions for entering remote control commands manually are also included.

For further information, see the function and parameter description in the "Properties" view within the QuickStep "Test Procedure Editor".

9.3.1.2 RS_RfSignalAnalyzerBase

This block provides a basic set of the most common remote control commands for the control of signal analyzers including commands to set up and read back traces. Standard functions like Init and Close and functions for entering remote control commands manually are also included.

For further information, see the function and parameter description in the "Properties" view within the QuickStep "Test Procedure Editor".

9.3.1.3 RS_NetworkAnalyzerBase

This block provides a basic set of the most common remote control commands for the control of network analyzers. Standard functions like Init and Close and functions for entering remote control commands manually are also included.

For further information, see the function and parameter description in the "Properties" view within the QuickStep "Test Procedure Editor".

9.3.1.4 RS_PowerSupplyBase

This block controls a power supply with respect to DC voltage and current settings and with respect to current measurements. A set of remote control commands is provided for this purpose. Standard functions like Init and Close and functions for entering remote control commands manually are also included.

Characteristic setting actions and functions:

- Select the type of power supply, command the power supply to provide an initial DC voltage and current for the DUT (function: Init).
- Define the DC voltage and current provided for the DUT.
- Enable measuring the current from the DUT.

For further information, see the function and parameter description in the "Properties" view within the QuickStep "Test Procedure Editor".

9.3.1.5 RS_RFFEBase

This block controls theScout SC4410 USB-to-RFFE device used for MIPI RFFE communication with the DUT.

Characteristic setting actions and functions:

- Enable RFFE HYPERLINK \l "page191" MIPI communication and set the input/output voltage used fort his communication (function: Init).
- Define RFFE commands or select a file with RFFE commands (function: Execute RFFE Commands).

Currently the basic support for the following types of power supply is included: R&S NGMO, Keysight E364x, Keysight N6700.

For further information, see the function and parameter description in the "Properties" view within the QuickStep "Test Procedure Editor".

9.3.1.6 RS_PowerSensorBase

This block controls R&S power sensors like NRP-Zxx, NRPxxS(N) or NRPM3 and provides the power measurement results of the sensor. A set of remote control commands is provided for this purpose. Standard functions like Init and Close and functions for entering remote control commands manually are also included.

For further information, see the function and parameter description in the "Properties" view within the QuickStep "Test Procedure Editor".

9.3.1.7 RS_OscilloscopeBase

This block controls an R&S RTO Oscilloscope. This block provides a basic set of the most common remote control commands for the control of oscilloscopes. Standard functions like Init and Close and functions for entering remote control commands manually are also included. For further information, see the function and parameter description in the "Properties" view within the QuickStep "Test Procedure Editor".

9.3.1.8 RS_OSPBase

This block controls an R&S OSP, the Open Switch and Control Platform. The functions provide means for defining and switching RF paths as well as for handling configurations and relays. For details, see the R&S OSP Operating Manual.

For further information, see the function and parameter description in the "Properties" view within the QuickStep "Test Procedure Editor".

9.3.1.9 RS_Positioner

This block controls an antenna positioner of type ATS-CCP1, F100 or F200 pan/tilt. The functions provide means for configuring the position parameters (for example elevation, azimuth, speed) and more.

Init
Selects the antenna positioner type and establishes a VISA connection to the Positioner.

In case of an ATS-CCP1 positioner you can additionally set factory/mechanical angle and speed limits. The limits in this function are used for generating accurate error messages if the positions or speeds requested by other functions are out of the allowed range.

"Positioner Model"	Select "F100", "F200" or "ATS-CCP1".
"VISA Resource"	Enter the VISA string for the VISA connection.

PrintIdentity
Requests the identity string of the connected positioner and writes it to the execution protocol and log viewer.

Close
Closes the VISA connection to the positioner.

Gui
Opens a graphical user interface to set the position and speed of the positioner in manual operation.

Read Position
Returns the current angle of the selected axis.

A constant position offset set with the "Set Position Offset" function is taken into account.

Set Position
Sets the angle for the selected rotation axis.

Note that if the requested position exceeds the factory/mechanical limits of the positioner, the positioner will not move.

"Axis"	Select "Azimuth angle (phi)" or "Elevation (theta)".
"Position"	Set the angle value in degrees for the selected axis.
"Mode"	Select "Absolute" or "Offset". With "Offset" selected the configured angle value is applied to all "Set Position" and "Read Position" actions that use absolute angle values.

Set Position Offset
Sets a constant angle offset for the selected rotation axis. The offset is applied to all "Set Position" and "Read Position" actions with absolute angle values.

"Axis"	Select "Azimuth angle (phi)" or "Elevation (theta)".

"Offset" Set the angle value in degrees for the selected axis.

Wait for Position

Delays the next test procedure step until the previous positioning request has been finished.

F100 positioner: The function uses the positioner's "Await" command.

ATS-CCP1 positioner: The function polls the current position and waits until the requested position is reached within +/-0.1 degrees for each rotation axis.

Set all Axis and Wait

Sets the azimuth and elevation in absolute values and waits until that position has been reached.

This function is mainly a combination of the functions "Set Position" and "Wait for Position". An "Additional Delay" is applied after "Wait for Position" has been finished.

Reset

Sets the position to azimuth 0° and elevation 0°.

For the F100, a full reset calibration is executed. This process takes several seconds during which the positioner rotates over the full angle range for both axis ("Homing").

Set Limits

Confines the angle range for the selected rotation axis. The limit values set in this function have to be within the factory or mechanical limits of the used positioner.

F100 positioner: Azimuth limits have to be within -159°..159°, elevation limits have to be within -47°..31°. Disable the azimuth limits to achieve a -180°..180° azimuth range. Disable the elevation limits to achieve a -80°..31° elevation range.

"Axis" Select "Azimuth angle (phi)" or "Elevation (theta)".

"Limit Low" Set the minimum angle.

"Limit High" Set the maximum angle.

"Enable" Select "true"/"false" to apply/deactivate the limits.

Set all Axis Limits

Sets the azimuth and elevation limits in degrees. The limit values have to be within the factory or mechanical limits of the used positioner.

Set Acceleration

Sets an acceleration for the selected axis in degrees per square second.

The degrees are internally converted to steps of motor positions. The actual speed depends on the device and selected axis due to different resolutions and acceleration.

For a Tilt-Tilt positioner: Enter an acceleration value in the 300..600 range.

Set Speed

Sets the rotation speed for the selected axis. If the configured speed exceeds the maximum speed of the positioner, the configured speed is not applied.

"Axis"	Select "Azimuth angle (phi)" or "Elevation (theta)".
"Speed"	Set the speed in degrees per second for the selected axis.
"Mode"	Select "Absolute" or "Relative". With "Relative" selected the configured value is added to the previous value.

Halt

Immediatly decelerates and halts the rotation around the selected axis.

"Axis"	Select "Azimuth angle (phi)", "Elevation (theta)" or "All".

Step Mode

Only relevant for an F100/F200 positioner. Defines the tradeoff between moving speed and angle resolution for the selected axis.

A higher "Step Mode" ("Full", "Half") achieves a higher speed on cost of the resolution; a lower "Step Mode" ("Quarter", "Eighth") achieves a higher resolution on cost of the speed.

In "Auto" mode, the controller selects the correct step mode based on the current speed and processes all units as though the unit were in eighth step mode while allowing to move at the higher speeds available for the half step mode.

"Step Mode" values and resolution for an F100/F200 positioner (both axes):

- "Full": ca. 0.103 degree/step
- "Half": ca. 0.051 degree/step
- "Quarter": ca. 0.026 degree/step
- "Eighth": ca. 0.013 degree/step

Step Trigger
Only relevant for an ATS-CCP1 positioner. Enables or disables the hardware trigger and sets an angle interval in degrees. If the trigger is enabled, a trigger signal will be generated whenever the angle increase/decrease during rotation reaches the configured interval.

Write Command
Sends a manually entered SCPI command to the positioner.

Read Command
Fetches the reply for a previously sent "Write Command".

Query Command
Sends a manually entered SCPI command to the device and receives the reply. This function is a combination of the "Write Command" and "Read Command" functions.

9.3.2 Other Blocks

Blocks in this chapter complement functions of instrument blocks and provide device-independent functionality.

9.3.2.1 RS_UtilityBase

This block allows to modify a test procedure with some general helper functions.

Provided Functions

VISA Control
Specifies a SCPI command and sends it to a target instrument determined by its VISA connection (containing the target address).

"Command Type"	"Write", "Read", "Query" can be selected.
"VISA Resource"	The VISA address string to the target instrument. An IP or GPIB address is expected.
"Remote API"	The interface or protocol for connecting the VISA application layer with the TCP transport layer. For more details see "Init Parameters" on page 287.

"SCPI Command"	The string defining the actual SCPI command. Only used for the command types "Write" and "Query".

Instrument Show Screen

Establishes a remote desktop or remote LXI connection to a test instrument. The instrument's display is shown in an extra window and the instrument can be controlled remotely from this window.

"Connection Type"	Select "Remote Desktop Connection" or "Remote LXI Connection". "Remote LXI Connection" is used for a connection over a web interface with the test instrument running an LXI web server.
"Server"	Specifies the VISA resource string for the connection.
"User Name, Password"	The credentials for the login on the test instrument. The entries are not used, if no credentials are needed at the test instrument.
"Window Name"	The title of the window and also the identifier to be used by the "Instrument Close Screen" and "Instrument Window Shot" block functions.
"Monitor"	Determines which monitor displays the preview if two monitors are used.
"Window Height, Window Width"	Defines the hight and width of the preview window in percent of the display size.
"Y-Positioning, X-Positioning"	Defines the position of the top-left corner of the preview window. 0% means top or left, 100% means bottom or right.

Instrument Close Screen

Closes a remote connection window (or several ones if required).

"Close All Windows"	If activated ("True"), all opened remote connection windows are closed. If deactivated, the "Window Name" variable becomes relevant.
"Window Name"	Closes the remote connection window with the specified name.

Instrument Window Shot

Takes a screenshot of a remote connection window and stores it as `.bmp` file.

"Window Name"	The name of the remote connection window which the screenshot is taken from.

"Path" Specifies the path and name of the screenshot file. The default path (no entry) directs to the results folder.

Show MessageBox

Defines the type and content of a message box displayed during testrun when the block function is executed.

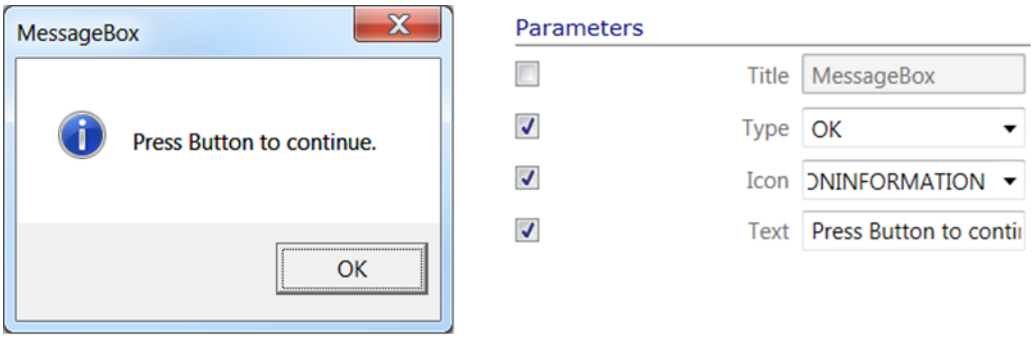


Figure 9-41: Message box

"Title" The headline of the message box.

"Type" Specifies the buttons of the message box and thereby the type of available interactions.

"Icon" The icon indicating the information type of the message box: "Error", "Question", "Warning", "Information".

"Text" The text displayed in the message box.

Show Image

Provides a pop-up window with user definable picture.

Delay

Halts the execution of the subsequent connected block function in the test procedure for a specified time.

"Unit" "Microseconds", Milliseconds and "Seconds" can be selected.

"Delay" A number which, combined with the unit, defines the halting time.

Get Path Loss

Calculates the path loss of a path defined in the system configurator at a given frequency and returns the value.

Set Project Variable

Sets or calculates an output value which is assigned to a test project variable via $G reference. The output value is either directly set by "Value" or is the result of modifying the "Input" value according to a selectable "Mode".

This function allows, for example, to increment a test project variable. In this case both the "Input" and the "Output" parameters address the same test project variable by a $G reference.

"Value" A value used for creating an output value to be assigned to the test project variable.

"Mode" Defines how the output value to be assigned to the test project variable is established:

- "Set": The output value is set to the content of "Value".
- "Increment": The "Input" value is incremented by "Value" and the result is the output value.
- "Decrement": The "Input" value is decremented by "Value" and the result is the output value.
- "Append": The "Value" is appended to "Input" and the result is the output value. This option is often useful for string variables.
- "Append with ;": The "Value" and a semicolon is appended to "Input" the result is the output value. This option is useful for variables of type double to create arrays.
- "Output": Defines by $G reference which target test project variable gets the output value resulting from the previous parameters. Enter the reference string $G.<Id> to the wanted test project variable.

"Input" A value used for creating an output value as defined with the "Mode". The value is usually fetched from a test project variable by $G reference.

"Output" Contains the $G reference to the target test project variable which gets the previously calculated output value.

9.3.2.2 RS_Report

This block allows to create and configure doc (Word), pdf and HTML reports as part of the test procedure. A preview of the report can also be displayed during runtime of the test.

Concept: See Reports

Provided Functions (Selection)

Init

Initializes the reporting functionality and offers a report preview displayed in a separate window during test run. This block function is mandatory and is typically defined the "Testrun Before" test execution phase.

"ShowPreview "	If activated ("True"), the report is displayed during test run in a preview window. Default: "False".
"Window Height, Window Width"	Defines the hight and width of the preview window in percent of the display size.
"Y-Positioning, X-Positioning"	Defines the position of the top-left corner of the preview window. 0% means top or left, 100% means bottom or right.
"Monitor"	Determines which monitor displays the preview if two monitors are used.

CreateReport

Creates a report file and determines the text for some elements in the header section of the report. If this block function is defined in the "DUT Loop Before" phase, a separate report is created for each DUT.

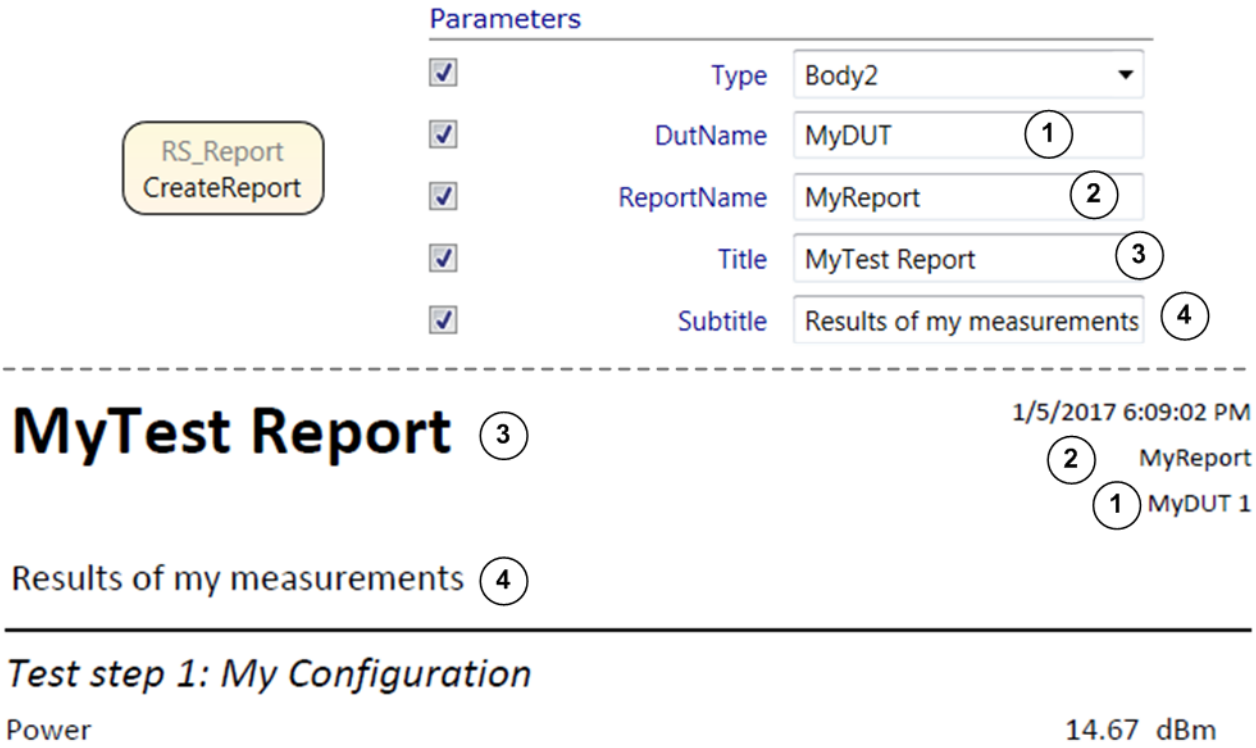


Figure 9-42: CreateReport

"Type"	Defines the appearance of the report header. Select a type from the list or enter path and name of an appropriate `.rdlx` file.
"DutName, ReportName, Title, Subtitle"	Defines the addressed elements of the report header.

AddMessage

Adds text in the result section of the report (under the header section).

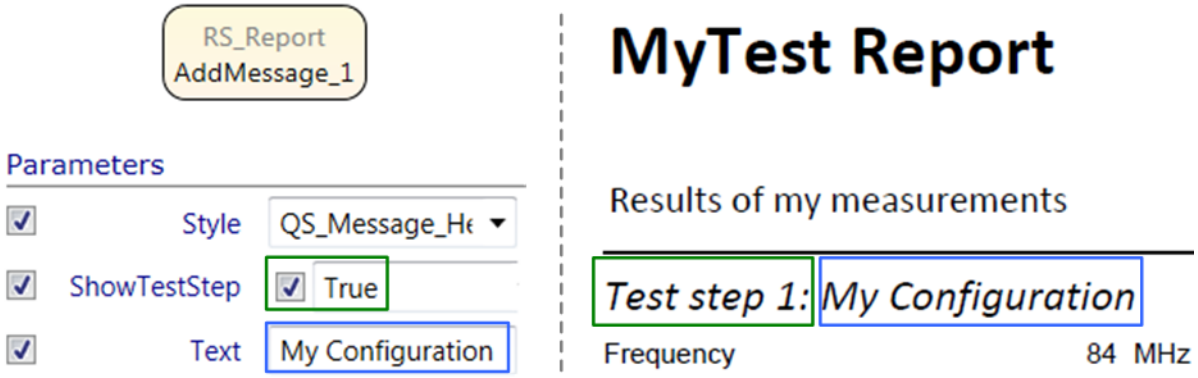


Figure 9-43: AddMessage

"Style"	Defines the appearance of the text in the "Text" variable (see below). Select a style from the list or enter the name of a style available in an appropriate `.rdly-styles` file.
"ShowTest-Step"	If activated, "Test step <current number>: " is inserted before the text in the "Text" variable.
"Text"	Enter the text to be added in the report.

AddResultMinMax

Adds one result line for a parameter in the report including a limit evaluation.

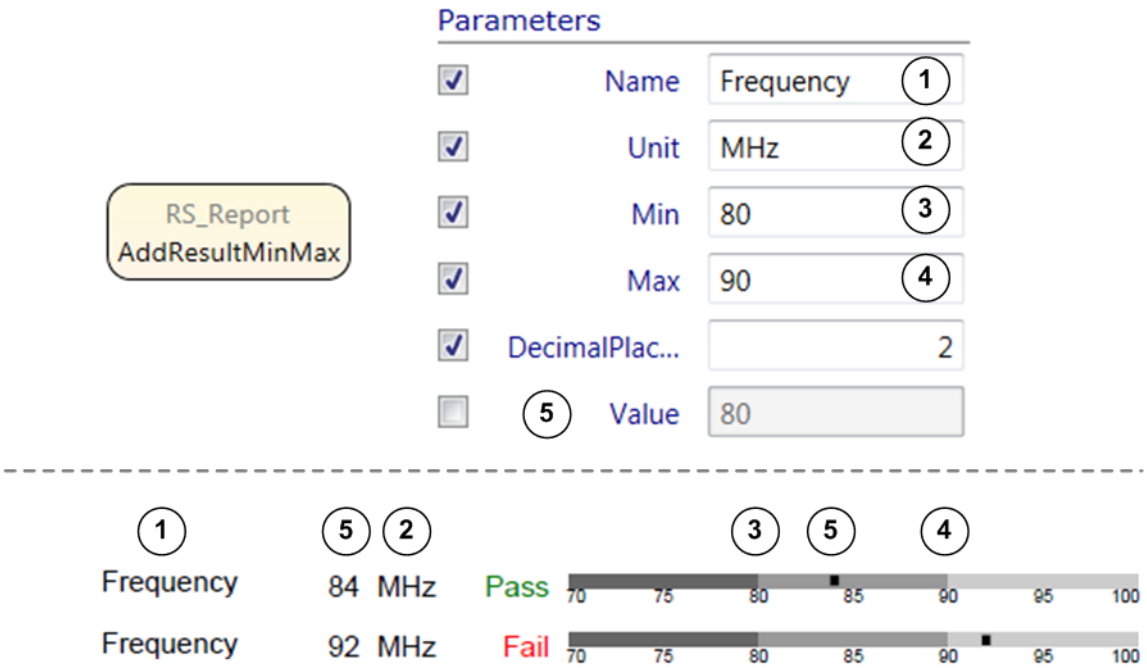


Figure 9-44: AddResultMinMax

"Name, Unit, Value"	These variables define the result components. The "Value" is usually unticked in the Test Procedure Editor and set by a *$R.* result reference in the test plan.
"Min, Max"	Limits for the result value. The report includes a value range bar showing the limits and the actual result value. "Pass" is displayed in the report if the result value is within the limits. Otherwise, "Fail" is displayed.
"DecimalPlaces"	Number of digits displayed for the result value.

AddResult

Adds one result line for a parameter and its value and unit in the report.

"Style"	Defines the appearance of the result line. Select a style from the list or enter the name of a style available in an appropriate `.rdly-styles` file.
"ResultName, Unit, Format, ResultValue"	These variables define the components of a result line. The "ResultValue" is usually unticked in the Test Procedure Editor and set by a *$R.* result reference in the test plan.
"DecimalPlaces"	Number of digits displayed for the result value.

CreateTable

Creates an empty table in a report.

"TableId"	A unique, identifying name used by other block functions to fill the table cells.
"ColumnCount"	Number of table columns (including a possible headline).
"RowCount"	Number of table rows.
"ColumnWidth"	The widths of the columns, each width in percent of the page width. Separate the widths with a semicolon and take care that the sum of them is 100, for example 25;25;25;25 in case of a table with four columns.

AddDataRowToTable

Adds data to a row in the specified table.

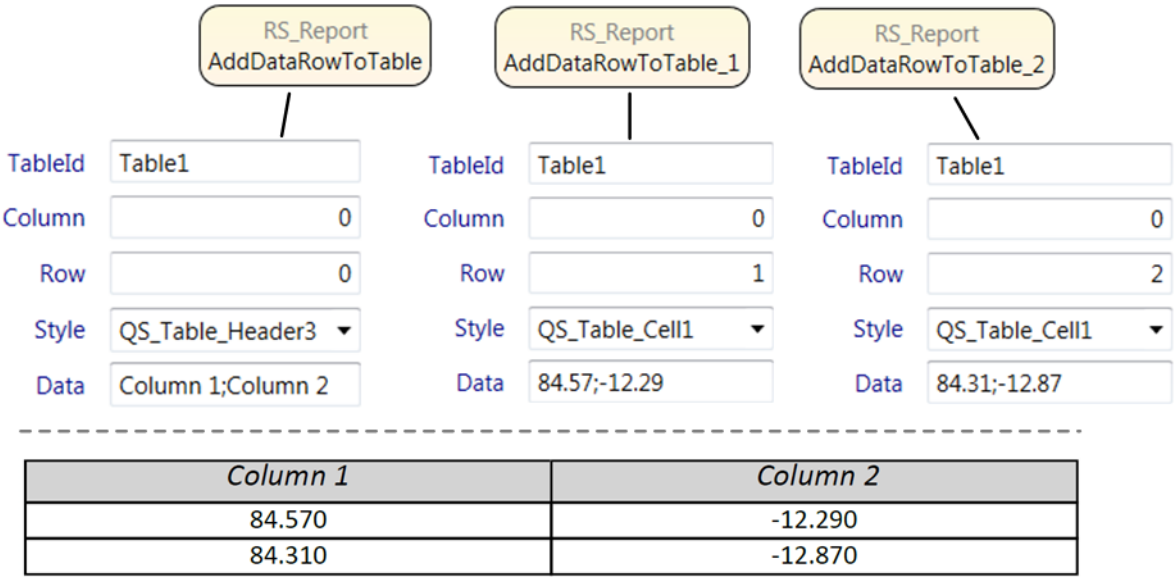


Figure 9-45: AddDataRowToTable (example)

"TableId"	Identifies the table which data is added to.
"Column"	The column where the adding of data starts. *0* denotes the first table column.
"Row"	The row where the adding of data starts. *0* denotes the first table row.
"Style"	Specifies the style of the cells in the row. Select from the list or enter a style which is included in an available `.rdly-styles` file.
"Data"	The data for the table cells of the specified row. Separate the entries for different cells by semicolon.

TraceToChart

Adds a chart with data from a trace file to the report.

"ChartType"	Select between "Cartesian", "Polar" and "Smith".
"ChartStyle"	Specifies the type of background (several light and dark options) and adjusts grid and font colors.
"LineStyle"	Select between a line connecting the data points or symbols for the data points.
"TraceFile"	Specifies the path and name of the trace file with the data for the chart.

AddSubreport

Adds a new subreport to the current report.

How to: See Setting Up a Report Including a Subreport

"SubreportId"	An identifying name (text string) used by AddResultToSubreport block functions to connect to the subreport and fill it with result data.
"Subreport-Path"	The path to the `.rdlx` subreport definition including the file-name.

9.3.2.3 RS_Visualization

This block provides functions to display results during testrun in a chart within an extra window.

Concept: See Chapter 4.11.1, "Charts", on page 66

Provided Functions (Selection)

CreateChartWindow

Creates an extra window and defines the properties of the chart to be displayed in the window.

"ChartName"	Specifies the title of the chart window.
"ChartStyle"	Specifies the type of background (several light and dark options) and adjusts grid and font colors.
"Chart Type"	Select between "Cartesian", "Polar", "Smith", "3D", "Histogram" and "Spectrogram".
"Window Height, Window Width"	Defines the hight and width of the chart window in percent of the display size.
"Y-Positioning, X-Positioning"	Defines the position of the top-left corner of the chart window. 0% means top or left, 100% means bottom or right.
"Monitor"	Determines which monitor displays the chart window if two monitors are used.
"Axis 1 Label, Axis 2 Label, Axis 3 Label"	The labels at the axis of the charts.

AddCurve...

Specifies the properties of the curve to be displayed in the chart and of the axes.

Note that several AddCurve... block functions can be used for one chart window.

"Curve ID"	A unique, identifying name used by other block functions to add data to the curve. Space characters in the entered string will be ignored during test execution.
"CurveStyle, CurveColor, Line Width"	Specifies how the curve is displayed.
"Fraction Number"	The number of digits behind the "." used for displaying the current Y-value (left from the chart).
"Y-Value Unit (AddCurvePolar: Radial-Value Unit)"	The unit displayed at the Y-axis (AddCurvePolar: Unit in radial direction).

"Axis Type, Axis ... Scale Type"	Specifes axis properties.
"Marker"	If activated, adds a marker to the curve. The marker can be moved along the curve and displays the current coordinates.

AddDataToCurve
Specifies the data points for the curve and draws the curve according to the specified curve ID.

"Curve ID"	Identifies the block function instance defining the curve properties. Space characters in the entered string will be ignored during test execution.
"Axis 1 Value, Axis 2 Value"	Specifies the data points for the curve.
"Axis 1 selection"	If activated, automatically generated axis 1 values can be used: "Teststep Number" or "Autoincrement". The value "Use Axis 1 Parameter" indicates that the axis 1 values are determined by the "Axis 1 Value" variable.
"History Buffer"	Number of values building the curve. If the number of recorded values exceeds this number, earlier values are thrown out (first in, first out). Use the value "0" to disable the buffer: all values are included in the curve; consequenly, the curve is compressed if the number of recorded values gets very high. Note that a non-zero history buffer value can be used to simulate an oscilloscope view.

9.3.2.4 DUT_Handling

This block allows to enter and log information about the used DUT(s) during test execution. Thereby, the measurement results can easily be related to the tested DUT(s).

Provided Functions

AskForDutInfo
Opens a dialog during test execution where the used DUT and the test situation can be characterized. In the function, the information elements of the dialog are represented as parameters whose values can be set.

The block functions should be located in the "DUT Loop Before" phase.

- If the check box for "Write to Execution Log" is activated, the DUT ID is stored in the execution log file.
- Clicking the "Test DUT" button resumes the testrun.
- If the AskForDutInfo function is called again (in the next repetition of the DUT Loop phase where the block function should be located), the dialog is re-opened and the DUT information of the previous AskForDutInfo calls is listed under "Tested DUTs".
- Clicking the "Stop Testing" button stops the testrun.

LogDutInfo
Selects the DUT-related information elements to be logged in the test run results. The results table in `TestStepResults.log` will contain columns for those information elements which are activated (via check box).

The same DUT information elements are available as in the "AskForDutInfo" function.

This block function should be located in the "Test Procedure" phase to have this information available in each line of the result log.

9.3.2.5 RS_Mathematics

This block provides functions for evaluating mathematical expressions:

- **Calculation**: Provides a set of basic calculation operations with two operands. This block function is intended for training purposes and as example for a software block.
- **MathExpression**: Evaluates expressions with up to three variables, mathematical operators (+, -, *, /, %, ^) and numerous functions. The evaluation is based on the C++ Mathematical Expression Toolkit Library (ExprTk).

9.3.2.6 RS_CallExeDll

This block provides functions for calling (executing) functions and executables. The functions may come from different sources, for example from DLLs.

Provided functions (selection)

Init
Initializes activities for the block. Particulary, defines the path to the DLL file containing the functions to be called by "CallFunction" or "CallMLFunction" block functions. The function is typically used in the "Testrun Before" phase.

AddDllSearchPath
Defines a path where a DLL file is searched for if it is not available in the usual directories. This function becomes important if a DLL file requires another, external DLL file under an unexpected directory. The path to this external DLL file must be specified. If the external DLL has dependencies to further DLLs, their paths must also be specified (more instances of the "AddDllSearchPath" block function).

CallFunction
Calls and executes a function from a DLL file. The path to the DLL file is defined by the "Init" block function beforehand. Also specifies the data types and values of input parameters for the DLL function if it expects parameters.

"DllFunction-Name"	Name of the called function in the DLL file.
"ReplyType"	Data type returned by the called function. Here, this parameter defines the data type of the Out (reply) parameter (whose value can be assigned to a test procedure variable).
"Para1Type"	The data type of the first parameter expected by the called function. Select "Void" if the called function requires no parameters.
"Para1"	The value of the first parameter expected by the called function.
"Para<n>Type"	The data type of the n-th parameter expected by the called function. Not used if a previous "Para<m>Type" (m < n) is set to "Void". Select "Void" if the called function expects only n - 1 parameters.
"Para<n>"	The value of the n-th parameter expected by the called function.

CallCSFunction

Calls a C# method from a class within an assembly (`.dll` or `.exe`). Also specifies the data types and values of input parameters if the method expects parameters.

"Assembly-Path"	The path to the assembly (`.dll` or `.exe`) containing the class and method to be called.
"FullClass-Name"	Name of the class containing the method to be called including the namespace.
"MethodName"	Name of the called method.
"ReplyType"	Data type returned by the called function. Here, this parameter defines the data type of the Out (reply) parameter (whose value can be assigned to a test procedure variable).
"Para1Type"	The data type of the first parameter expected by the called function. Select "Void" if the called function requires no parameters.
"Para1"	The value of the first parameter expected by the called function.
"Para<n>Type"	The data type of the n-th parameter expected by the called function. Not used if a previous "Para<m>Type" (m < n) is set to "Void". Select "Void" if the called function expects only n - 1 parameters.
"Para<n>"	The value of the n-th parameter expected by the called function.

CallMLFunction

Calls and executes a MATLAB script from a DLL file. The path to the DLL file is defined by the "Init" block function beforehand. Also specifies the data types and values of input parameters for the MATLAB script if it expects parameters.

"DllFunction-Name"	Name of the called function in the DLL file.
"Java Virtual Machine"	Activate this option if Java is used in the compiled MATLAB script.
"Para1Type"	The data type of the first parameter expected by the called function. Select "Void" if the called function requires no parameters.

"Para1" The value of the first parameter expected by the called function. For the parameter type "matrix", enter the matrix elements as shown in the figure (matrix elements of first row from left to right separated by ";", then "|" row separator, then matrix elements of second row from left to right separated by ";" and so on).

Input matrix for MATLAB script	Parameter „value“ in block function
1 7 5 2 4 3 9 6 8	1;7;5|2;4;3|9;6;8

Figure 9-46: Value string for a matrix

"Para<n>Type" The data type of the n-th parameter expected by the called function. Not used if a previous "Para<m>Type" (m < n) is set to "Void". Select "Void" if the called function expects only n - 1 parameters.

"Para<n>" The value of the n-th parameter expected by the called function.

CallExe

Calls an executable program (`.exe` file) with arguments (if expected) and executes it.

"PathToExeInclFile" Path and name of the executable program (`.exe` file).

"Arguments" Arguments as expected from the executable program. Separate the arguments by white space.

"Hidden" If selected, no command line window is opened to show the execution steps of the executable program.

"WaitUntilExit" If selected, QuickStep waits until the executable program is closed before continuing with the test plan execution. If not selected, the executable program runs in parallel.

"Read Std Output" If selected, the console output is logged and displayed in the Log Viewer.

9.3.3 Special Parameter Groups

9.3.3.1 Init Parameters

Init Parameters

This section mainly contains the parameters for establishing the control connection between a test instrument and the PC where QuickStep is running. The connection carries the SCPI commands in both directions. The test application uses VISA (Virtual Instrument Software Architecture) as standard interface providing input and output functions to communicate with the test instrument.

"Force Sending SCPI Commands"	If activated, the value of a device parameter required at a test instrument is transmitted to that instrument via SCPI command even if the last value of the parameter is still valid (no value change). Note that a device parameter keeps its last value on the test instrument, so a re-setting via SCPI command is usually not necessary if the last value can be re-used. Activating "Force Sending SCPI Commands" could slow down the test execution due to a high number of SCPI commands.
"Visa Resource"	The connection type, channel and address information of the instrument used for communication via SCPI commands. This is the so called VISA resource string. • Use the drop-down menu to select a reference to a predefined VISA alias. • Use the "Visa" button to create a VISA string. • Directly enter the VISA string.

Annex

A File Extensions and File Locations

File extensions

Extension	Meaning and Content
`.bdf`	Block definition file, contains the block definitions in XML format.
`.tpl`	Test plan file, the frame for test plan, test procedures, system definition and other components of a test.
`.sdf`	System definition file, contains the system definitions (for example devices and connections) in XML format. A `*.sdf` file is created when you export the current system definition in the "System Configurator".
`.log`	Log file, reports messages, states, errors, user defined text or other information as readable text.

File locations

The user defined projects and blocks as well as the OTA application project are usually stored under `C:\Users\Public\Documents\Rohde-Schwarz\QuickStep\` but the path depends on the Windows installation and the company IT rules. So the location of the public user directories might be different.

You can easily find the project folder via the QuickStep links in the start menu, see the figure.

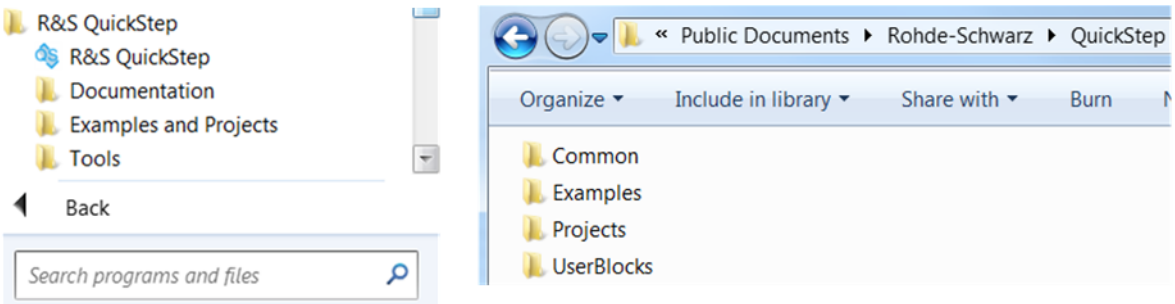


Figure A-1: Getting the project folder via Start menu

The QuickStep program files (GUI, application, block library etc.) are located under `C:\Program Files\Rohde-Schwarz\QuickStep\`.

B Return Codes in Command Line Mode

When QuickStep, operated in command line mode, ends operation with a program exit, a number ("return code") is returned indicating the exit cause. The following table describes the exit causes for the return codes. These descriptions are also available in the `ReturnCodes.h` file located in the installation directory, usually under
`C:\Program Files\Rohde-Schwarz\QuickStep\Framework\include`.

The return codes may help the user to react to a QuickStep exit. Further information, for example about the component which caused an error, is provided in the QuickStep ExecutionProtocol.

Table B-1: Return codes

Return code	Exit cause
Regular QuickStep exit, no error	
0	Successful test run
1	Limit failure
2	Test run aborted by user
3	Test run killed by user
Start errors	
-1	Input arguments have wrong syntax
-2	QuickStepEngine dll library could not be loaded
-3	LogModule dll library could not be loaded
-4	UserBlock dll library could not be loaded
-5	Method from dll could not be loaded (either from QuickStep dll or from UserBlock dll)
Errors before the testplan is executed	
-10	Test plan document could not be loaded. (Wrong path?)
-11	Test plan XML structure is not correct
-12	Check of required and provided functions failed
-13	CheckBlock failed
-14	BDF check failed
-15	DataModel load failed
-16	License check for QuickStepEngine.exe failed

Return code	Exit cause
-17	Block name of one or more blocks is too long
-18	Test plan does not contain any test step
Communication and connection errors	
-30	Number of sockets in the environment exceeds the limit. Maybe too many blocks
-31	Internal communication error occurred
-32	Connection to a user block timed out
Runtime errors of user blocks	
-40	Handled exception (BlockException) in a user block was thrown
Runtime errors of environment	
-50	Unhandled exception in a user block or system block was thrown Also: Any other exception in a user block or system block than a BlockException was thrown
-51	Unhandled exception in the QuickStep engine was thrown

C Supported Test Instruments

Blocks are provided to include the supported test instruments in test procedures and system configurations.

Generators:

- R&S SGT
- R&S SMW200A
- R&S SMBV100A

Analyzers:

- R&S FPS Signal Analyzer
- R&S FSW Signal Analyzer
- R&S FSV Signal Analyzer
- R&S ZNB Vector Network Analyzer

Power Supplies:

- R&S NGMO
- Agilent E3646A
- Agilent N6700B

Other devices:

- R&S NRP-Zxx, NRPxxS(N) Power Sensors
- Signal Craft SC4410 USB to RFFE Interface
- R&S OSP
- R&S RTO Oscilloscope
- R&S DUT: Evaluation board for advanced power amplifier testing

D MIPI RFFE Communication

The MIPI (Mobile Industry Processor Interface) alliance specification for RF Frontend Control Interface (RFFE) was developed to offer a common and widespread method for controlling RF frontend devices such as power amplifiers.

The Scout SC4410 (USB-to-GPIO/serial adapter) device from Signal Craft Technologies supports an RFFE-like interface for communication with MIPI RFFE devices. It can be used for example to enable/disable the DC power supply modulator and the power amplifying unit at the tested power amplifier via RFFE commands. Disabling these subcomponents in inactive times can reduce the overall power consumption of the DUT significantly.

MIPI RFFE commands

A MIPI RFFE command consists of a triple of data, each part given in hexadecimal format:

`<Slave address>:<Register address>:<Data>`, for example: `0x5:0x8:0x2E`.

- Slave address: The address of the target device getting the RFFE commands.
- Register address: The address of a register on the target device for storing a desired value.
- Data: Contains the value to be stored in the addressed register.

QuickStep supports (speaking) aliases to free the user from dealing with the triple hex format. The relation between aliases and the RFFE commands is provided in an XML file under `...\ConfigurationFiles\MipiFiles` folder of the test project. This file can be adjusted or a new file can be created.

Alternatively, lists of RFFE commands to be executed in a row can be set up in `*.csv` files in the same directory. Alias names are not supported in this case.

XML file for RFFE user commands

```
<?xml version="1.0" encoding="utf-8"?>
<RFFEUserCommands>
  <Commands>
  <Command Comment="Check Manufacturer ID"  CommandAlias="CheckManID"
               DeviceAddress="0x5" RegisterAddress="0x1E" value="0x00" />
  <Delay   Comment="Delay"            CommandAlias="DelayShort"      value="Delay:5" />
  <Delay   Comment="Delay"            CommandAlias="DelayLong"       value="Delay:50" />
  <Command Comment="EnvelopeTracking_On1"  CommandAlias="EnvelopeTracking_On1"
               DeviceAddress="0x5" RegisterAddress="0x1" value="0xB2" />
  <Command Comment="EnvelopeTracking_On2"  CommandAlias="EnvelopeTracking_On2"
               DeviceAddress="0x5" RegisterAddress="0x0" value="0x1F" />

      . . .

  </Commands>

  <Sequences>
    <Sequence Comment="ET_ON" SequenceAlias="ET_ON"
                CommandAliases="EnvelopeTracking_On1\Delay_ET\EnvelopeTracking_On2" />
  </Sequences>
</RFFEUserCommands>
```

Figure D-1: XML file for RFFE user commands (example)

In the XML file each "Command" has the following attributes:

- "Comment": Descriptive text, not used for program execution.
- "CommandAlias": Alias name usable in the test plan.
- "DeviceAddress": Slave address of the target device in hexadecimal format.
- "RegisterAddress": Register address at the target device in hexadecimal format.
- "value": Value to be written into the target register in hexadecimal format.

Commands can be sequenced under a unique sequence name to condense the test plan. Therefore, the "<Sequence>" XML tag has the following attributes:

- "Comment": Descriptive text, not used for program execution.
- "SequenceAlias": Alias name usable in the test plan.
- "CommandAliases": List of commands separated by "\".

The following special command is provided:

The command "Delay" defines a delay in multiples of the MIPI clock-cycle (13 MHz for the Scout SC4410 device). The "value" is given as "Delay:<NumberOfClockCycles>".

Sequence of RFFE commands in a *.csv file

The figure shows a `*.csv` file containing a list of RFFE commands. Each line contains one command and each command comprises a set of comma-separated entries.

```
comment1,comment2,0x0B,W,0x03,0x43,
comment1,comment2,0x0B,W,0x08,0x25,
comment1,comment2,0x0B,W,0x1A,0x08,
comment1,comment2,0x0B,W,0x05,0x41,
comment1,comment2,0x0B,W,0x1F,0x39,
comment1,comment2,0x0B,W,0x12,0x27,
```

1 2 3 4 5 6

Figure D-2: Command sequence in a *.csv file (example)

1 = Comment, not used in test execution
2 = Comment, not used in test execution
3 = Slave address
4 = Abbreviation for WRITE, or empty space
5 = Register address
6 = Value

Usage in test plans

If a test plan is connected with a test procedure that uses the block function "ScoutRFFE > Execute RFFE Commands", it displays the columns "Rffe Command" and "Rffe Files".

Rffe Command	Rffe Files
0x5:0x8:0x2E	File1
ET_ON\0x5:0x8	File1\File2\File

Figure D-3: Rffe columns in a test plan (example)

In the "Rffe Command" column, one can enter a command alias as defined in the XML file for RFFE user commands or the command as hexadecimal triple `<Slave address>:<Address>:<Data>`. Concatenation of several commands is possible using "\" as separator.

In the "Rffe Files" column, one can enter the file name (**without extension!**) of a `*.csv` file containing a list of RFFE commands as comma-separated values. Concatenation of several files is possible using "\" as separator.

If there are entries in both columns, the "Rffe Command"s are executed before the commands in the "Rffe Files".

D.1 Installing the SignalCraft Scout Driver

Install this driver only if the SignalCraft Scout USB to serial / GPIO adapter is used for controlling a DUT via MIPI-RFFE commands.

Starting situation: The SignalCraft Scout USB to serial / GPIO adapter has been connected to the PC and you are informed that the "Device driver software was not successfully installed".

1. Open the Windows "Device Manager" via the Windows "Start" button, e.g. by typing *device manager* into the "Search programs and files" input field.
2. Locate "Gadget Serial v2.4" under "Other devices" and right-click the entry to open the context menu.

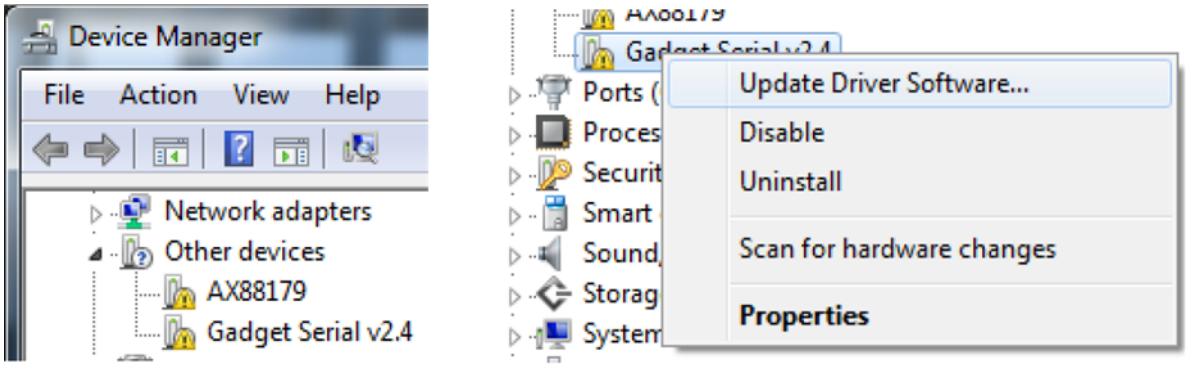


Figure D-4: Scout driver, Gadget Serial ...

3. Select "Update Driver Software…" to open the corresponding dialog.

Figure D-5: Scout driver, Update Driver Software dialog

4. Select "Browse my computer for driver software" and browse to the `<INSTALLDIR>\Firmware\` folder.
5. If a "Windows Security" warning appears, select "Install this driver software anyway".

Results

Successful installation is reported.

The Device Manager displays SignalCraft Scout in the "Ports" folder.

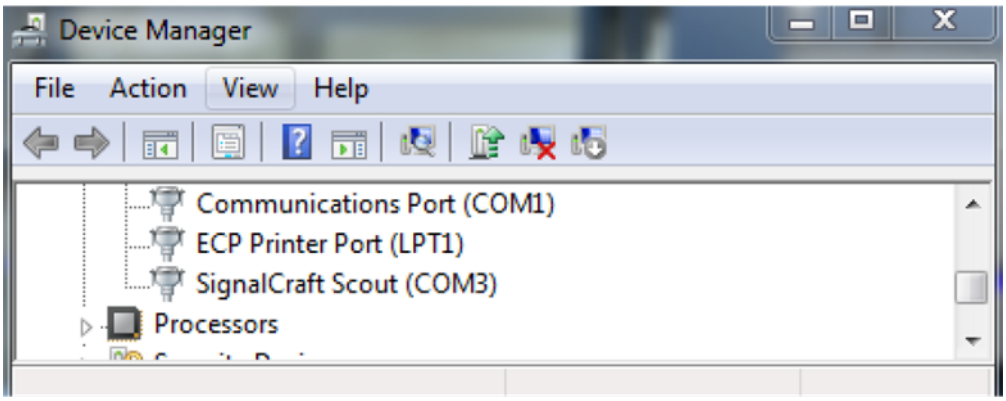


Figure D-6: Scout driver entry within the device manager

E Control Interfaces and Protocols

The following figure shows the protocol structure for controlling test instruments via SCPI commands.

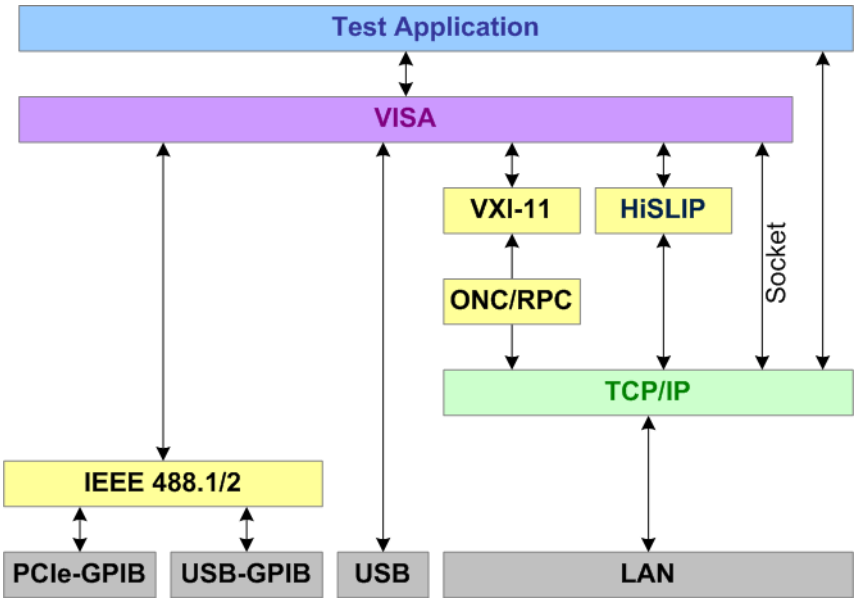


Figure E-1: Protocols used for controlling test instruments

TCP/IP ensures connection-oriented communication, where the order of the exchanged messages is adhered to and interrupted links are identified. With this protocol, messages cannot be lost.

SCPI

SCPI commands (Standard Commands for Programmable Instruments) are used for remote control of the test instruments. The SCPI standard is based on the IEEE 488.2 standard and aims at the standardization of instrument-specific commands, error handling and the status registers.

VISA

VISA (Virtual Instrument Software Architecture) is a standardized software interface library providing input and output functions to communicate with instruments. High level programming platforms use VISA as an intermediate abstraction layer. VISA encapsulates the low-level function calls and thus makes the transport interface transparent for the user.

HiSLIP

HiSLIP (High Speed LAN Instrument Protocol) uses two TCP sockets for a single connection, one for fast data transfer, the other for non-sequential control commands. The performance is as high as with raw socket network connections. HiSLIP supports features like message exchange protocol, serial poll and service request compatible with IEEE 488.2.

VXI-11

The VXI-11 standard is based on the ONC and RPC (Open Network Computing, Remote Procedure Call) protocols which in turn rely on TCP/IP at the transport/network layer. The TCP/IP protocol and the associated network services are preconfigured.

Raw socket

In raw socket mode, VISA connects directly with the TCP transport layer, so that the test application directly communicates via the network connection. Raw socket communication has small protocol overhead but is susceptible to communication errors.

Glossary: Abbreviations and Terms

A

ACLR: Adjacent Channel Leakage Ratio

AM-AM: Amplitude modulation before power amplification and amplitude modulation again after power amplification

AM-PM: Amplitude modulation before power amplification and phase modulation after power amplification

API: Application Programming Interface

B

BB: Baseband

BDF: Block Definition File

C

CHM: Compiled HTML Help or Compiled Help Module

CPP: C++

CSV: Character Separated Values

D

DC: Direct Current

DLL: Dynamic Link Library

DPD: Digital Predistortion

DUT: Device Under Test

E

EIRP: Equivalent Isotropically Radiated Power

ET: Envelope Tracking

EUTRA: Evolved UMTS Terrestrial Radio Access

EVM: Error Vector Magnitude

G

GLORIS: Global Rohde & Schwarz Information System

GPIB: General Purpose Interface Bus

GUI: Graphical User Interface

H

HiSLIP: High Speed LAN Instrument Protocol supporting message exchange protocol, serial poll and service request compatible with IEEE 488.2. HiSLIP creates two TCP connections from the control computer to the test instrument: one for data, the other for non-sequential control commands.

I

Icc: Current provided by a DC power supply

ID: Identifier

IDE: Integrated Development Environment

IP: Internet Protocol

iPAE: Instantaneous Power Added Efficiency

L

LAN: Local Area Network

LTE: Long-Term Evolution

LXI: LAN eXtensions for Instrumentation

M

MIPI: Mobile Industry Processor Interface specified by the MIPI alliance specification, a common and widespread interface for controlling RF frontend devices

N

NI: National Instruments

O

OTA: Over The Air

P

PA: Power Amplifier

PAE: Power Added Efficiency

PC: Personal Computer

PDF: Portable Document Format

Pout: Output power

PRC: Procedure

Q

QS: QuickStep

R

Raw EVM: Error Vector Magnitude compared to the reference signal

Raw Socket: Direct communication between VISA and the TCP transport layer, so that the test application directly communicates via the network connection. No support of asynchronous events like Service Requests (SRQs).

RDL: Report Definition Language

RegEx: Regular Expression. A sequence of characters defining a pattern which matching text need to conform to

RF: Radio Frequency

RFFE: RF Frontend

ROM: Read-only Memory

RS: Rohde & Schwarz

RX: Receive

S

Scout: SignalCraft Scout USB to serial/GPIO adapter. The adapter is used for controlling the DUT via MIPI-RFFE commands.

SCPI: Standard Commands for Programmable Instruments. SCPI commands are used for remote control of the test instruments. The SCPI standard is based on the IEEE 488.2 standard.

SDF: System Definition File

SEH: Structured Exception Handling

SnP: A Touchstone file. The file fomat is to be used for correction of frequency dependent attenuation between generator and DUT. Note: The attenuation values from the "Input SnP File" are added to the "Input Attenuation" value.

SRQ: Service Request

T

TCP: Transmission Control Protocol

TPL: Test Plan

TPR: Test Project

TRP: Total Radiation Power

TX: Transmit

U

UE: User Equipment

USB: Universal Serial Bus

V

Vcc: Voltage provided by a DC power supply

VIO: Voltage Input/Output

VISA: Virtual Instrument Software Architecture. VISA is a standardized software interface library providing input and output functions to communicate with instruments.

VNA: Vector Network Analyzer

VXI-11: A communication protocol specifically developed for controlling test and measurement instruments over TCP/IP and LAN. The VXI-11 standard is based on the ONC and RPC (Open Network Computing, Remote Procedure Call) protocols which in turn rely on TCP/IP at the transport/network layer. The TCP/IP protocol and the associated network services are preconfigured.

W

WCDMA: Wideband Code Division Multiple Access

X

XML: Extensible Markup Language

Index

P

R

S