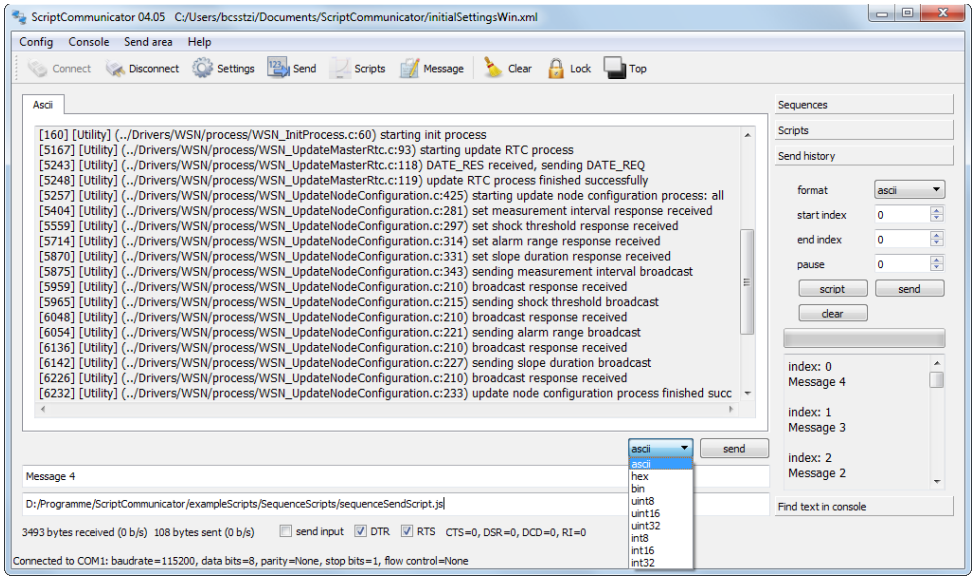
ScriptCommunicator

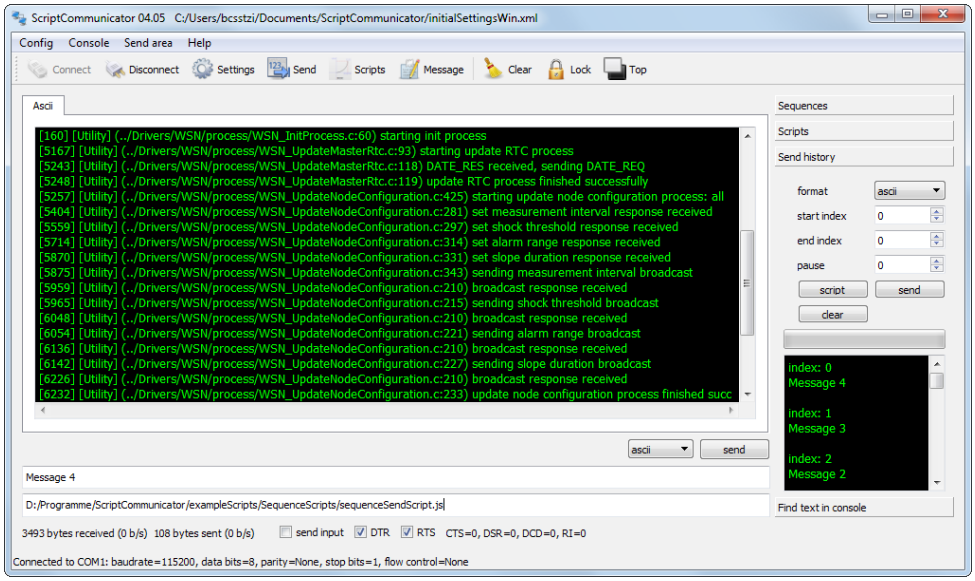
Manual

Table of contents

ScriptCommunicator (https://sourceforge.net/projects/scriptcommunicator/, https://github.com/szieke/ScriptCommunicator_serial-terminal) is a scriptable cross-platform data terminal which supports following interfaces:

- Serial port (RS232, USB to serial)
- UDP
- TCP client/server (network proxy support for TCP clients)
- SPI master (cheetah SPI)
- CAN (PCAN-USB, only on windows)



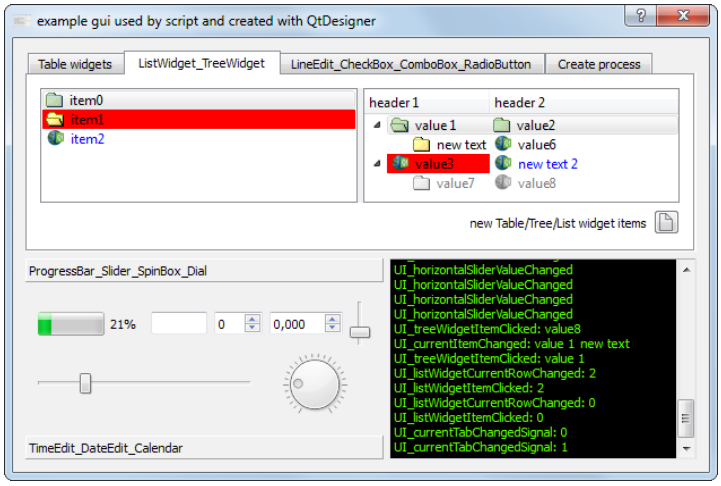


All sent and received data can be shown in a console and can be logged in an html, a text and a custom log.

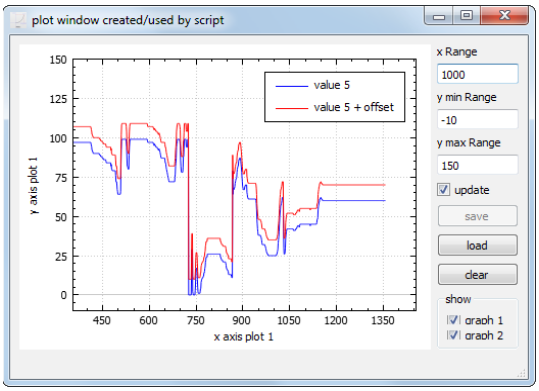
In addition to the simple sending and receiving of data the ScriptCommunicator has a script interface (QtScript). QtScript is based on the ECMAScript scripting language, as defined in standard ECMA-262. Microsoft's JScript, and Netscape's JavaScript are also based on the ECMAScript standard. For an overview of ECMAScript, see the ECMAScript reference.

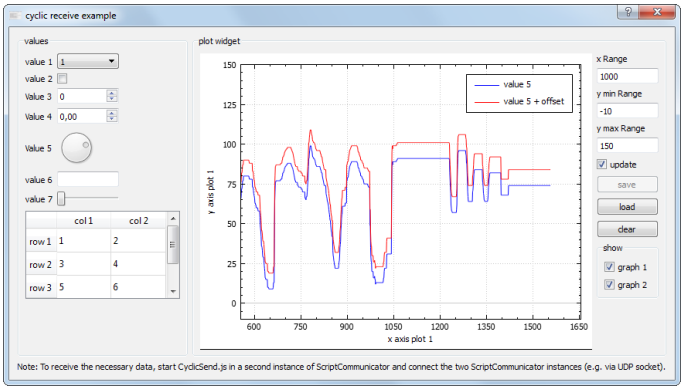
This script interface has following features:

- Scripts can send and receive data with the main interface.
- In addition to the main interface scripts can create and use own interfaces (serial port (RS232, USB to serial), UDP, TCP client/server and SPI master (cheetah SPI)).
- Scripts can be connected to GUI files which have been created with the QtDesigner or QtCreator. All elements in the GUI (files) can be accessed from the script.



- Multiple plot windows and plot widgets can be created by scripts (QCustomPlot from Emanuel Eichhammer is used).





- Dynamic link libraries with a special interface (see chapter Dynamic link libraries) can be loaded by script. Script function can be called by the library and vice versa.

A video which demonstrates the basic features of ScriptCommunicator can be found here:
https://www.youtube.com/playlist?list=PLniMuy2Q_xGuFB_kl1nte2mDxfeeOu8ce

ScriptCommunicator History

Revision	Date	Changes

02.00	2014-12-31	• PCAN interface added (only on windows)
		• command-line mode added
		• worker script, changed behavior:
		  ◦ loadUserInterfaceFile
		  ◦ renameDirectory
		  ◦ renameFile
		• worker script functions added:
		  ◦ isConnectedWithCan, canMessageReceivedSignal, createPcanInterface, sendCanMessage
		  ◦ showMultiLineTextInputDialog, showGetItemDialog,showGetIntDialog, showGetDoubleDialog
		  ◦ disconnect, connectPcan, connectSocket, connectSerialPort, connectCheetahSpi
		  ◦ showReceivedDataInConsoles, showTransmitDataInConsoles
		  ◦ addMessageToLogAndConsoles
		• ScriptWidget functions added:
		  ◦ setAdditionalData
		  ◦ getAdditionalData
		• ScriptTableWidget function added:
		  ◦ insertWidget
		• worker script, default parameter added:
		  ◦ readFile, readBinaryFile, readDirectory, checkFileExists, checkDirectoryExists, deleteFile, deleteDirectory, deleteDirectoryRecursively, createDirectory, loadLibrary,loadScript, sendDataArray, sendString

2.01	2015-01-05	• bug fixes: ◦ received CAN ids have been displayed not correctly
2.02	2015-01-18	• command-line mode: The script window is now invisible per default (it can be made visible by the command line argument -withScriptWindow) • send window: remove script menu added • ScriptWidget function added: ◦ blockSignals • ScriptSplitter added
2.03	2015-01-30	• bug fixes: ◦ writeFile and writeBinaryFile: calling this functions with the argument replaceFile=true did not replace an existing file • the number of sent bytes are now displayed in the main window • the console-colors are adjustable now

2.04	2015-02-05	• ScriptSpinBox functions added:
		  ◦ value, setSingleStep, singleStep
		• ScriptTabWidget signal added:
		  ◦ currentTabChangedSignal
		• ScriptTableWidget functions added:
		  ◦ verticalScrollBarWidth, isVerticalScrollBarVisible
		• script widgets added:
		  ◦ ScriptDoubleSpinBox, ScriptToolBox, ScriptDial, ScriptCalendarWidget, ScriptDateTimeEdit

2.05	2015-02-07	• internal improvements
		• ScriptSlider function added:
		  ◦ value
		• ScriptDial function added:
		  ◦ value

2.06	2015-02-13	• internal improvements
		• text console's wrapping mode changed
		• new console/log options:
		  ◦ new line after x sent/received bytes
		  ◦ new line after x milliseconds without sending or receiving data
		• the sequence table is now disabled during sending of data with the send window
		• the RTS and the DTR pin (serial port) can manually be set/cleared now (check boxes in the main window)
		• worker script function added:
		  ◦ setSerialPortPins
		• ScriptSerialPort function added:
		  ◦ setDTR, setRTS

2.07	2015-02-28	• Bug fix: no time stamps after running ScriptCommunicator longer then a day • all local IPv6 and IPv4 addresses are displayed in the socket tab (settings dialog) now
2.08	2015-03-02	• Console send mode: if the send input check box in the main window is checked all text entered in a console will be sent • ScriptTableWidget function added: ◦ setCellIcon • Bug fix: error while changing the format of a sequence in the send window

2.09	2015-03-05	• sequence script functions added:
		  ◦ calculateCrc8, calculateCrc16, calculateCrc32, calculateCrc64
		• new console/log option:
		  ◦ new line at byte (CR, LF, none and custom)
		• when the serial port combo box is clicked (settings dialog) a serial port scan is performed now

2.10	2015-03-13	• Bug fixes:
		  ◦ error while parsing a sequence or script config file with no entries
		  ◦ moving a window while a script shows/opens a dialog caused a complete freeze of ScriptCommunicator

2.11	2015-03-27	• improved console speed • new line at byte (CR, LF, none and custom) can be adjusted for the log and the console separately now
2.12	2015-04-10	• socket address input filter removed (no IPv6 addresses could be entered) • bug fix: sporadically crash after receiving a 'new line' character

2.13	2015-05-01	• bug fixes:
		  ◦ scriptThread.calculateCrc64 did not work correctly
		  ◦ uncaught exceptions in script functions connected to a signal have been ignored
		• ScriptComboBox functions added:
		  ◦ count, clear
		• ScriptTable function added:
		  ◦ rowsCanBeMovedByUser

2.14	2015-05-31	• fixed some issues in the manual • internal improvements

2.15	2015-06-12	• Script- and sequence-buttons added in the main window
		• Copy button added in the send window
		• worker and sequence script functions added:
		  ◦ setGlobalString, getGlobalString
		  ◦ setGlobalDataArray, getGlobalDataArray
		  ◦ setGlobalUnsignedNumber, getGlobalUnsignedNumber
		  ◦ setGlobalSignedNumber, getGlobalSignedNumber

		• renamed the sequence script main object: table->seq • sent on enter added (settings dialog->console tab) • the console/log time-stamp format can be adjusted now • new default argument in scriptThread.appendTextToConsole
2.16	2015-06-14	• bug fix: Crash while ScriptCommunicator is closed and a script is running
3.00	2015-06-19	• custom console and custom log added • new default argument in scriptThread.addMessageToLogAndConsoles • new menu in the send window: edit sequence script
3.01	2015-06-24	• new manual chapter added: Internal architecture • new custom console example (exampleScripts\CustomLogConsoleScripts\CustomConsole_QRCode) • new main window menus added: ◦ Console/print console, Console/save console ◦ Help/report a bug, Help/feature request
3.02	2015-07-02	• bug fix: missing Qt libraries for printing a console
3.03	2015-07-04	• fixed some issues in the manual • added a custom build of QtDesigner (only the supported widgets/GUI elements (supported by ScriptCommunicator) are available now)
3.04	2015-07-31	• the log and the console tab (settings dialog) have been redesigned • internal improvements • fixed some issues in the manual
3.05	2015-08-02	• the console mixed mode utilize the width of the console better now • bug fix: save console crashed on Linux if no file ending has been given
3.06	2015-08-08	• main window: ◦ find text in console area added ◦ the send and the receive data rates are now displayed • sending data with the main interface is faster now • new example worker scripts: ◦ WorkerScripts\CyclicSendReceive\CyclicReceive.js ◦ WorkerScripts\CyclicSendReceive\CyclicSend.js • worker script function added: setScriptThreadPriority
3.07	2015-08-13	• new script widget: ◦ ScriptPlotWidget • ScriptPlotWindow: ◦ functions added: clearGraphs, removeAllGraphs, showHelperElements, clearButtonPressedSignal ◦ function removed: showFromScript (use show instead) • ScriptGroupBox function added: addPlotWidget • the CAN tab has been improved
3.08	2015-09-12	• bug fix: possible crash while stopping a script • TCP client: connection errors are now displayed • ScriptTcpClient functions/signals added: ◦ errorSignal, getErrorString • new main window menus added: ◦ Help/video, Help/get support
3.09	2015-09-12	• worker script functions added: ◦ getCurrentVersion • custom log/console and sequence script function added: getCurrentVersion • ScriptTcpClient, ScriptUdpClient and ScriptSerialPort functions added: enableMainInterfaceRouting, disableMainInterfaceRouting
3.10	2015-09-26	• network proxy support implemented (only for TCP clients) • ScriptTcpClient function added: setProxy
3.11	2015-10-04	• bug fix: exceptions in the script function 'stopScript' are displayed now • worker script function added: getFileSize • new arguments in worker script functions:

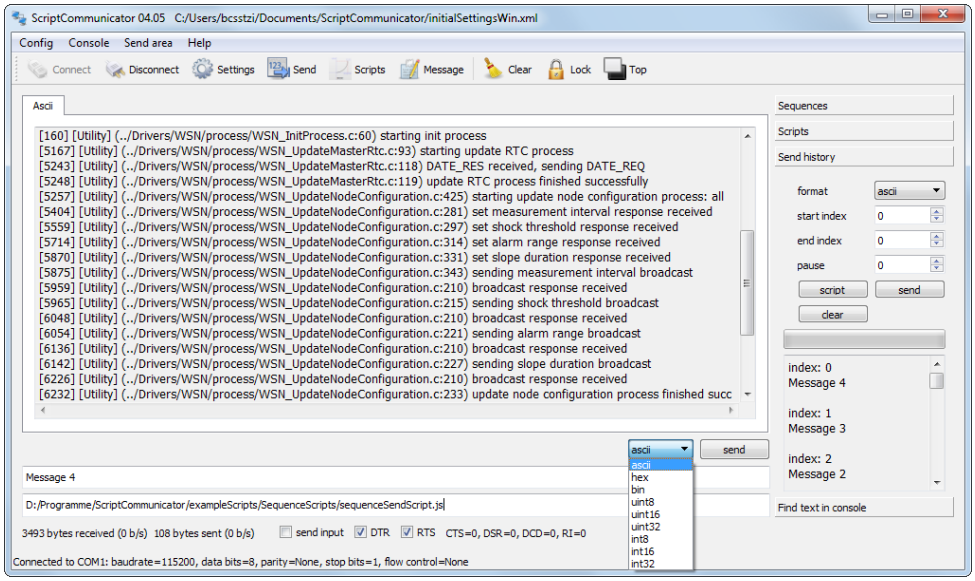
		◦ readFile, readBinaryFile, writeFile, writeBinaryFile • ScriptPlotWidget and ScriptPlotWindow function added/changed: ◦ load button, showHelperElements
3.12	2015-10-15	• console and log time stamp format extended (now date and time can be created) • new option 'time stamp after byte' added
3.13	2015-10-18	• command-line option '-notMinimized' added • worker script function added: exitScriptCommunicator • the template scripts have been modified • manual chapter 'Worker scripts' has been modified
3.14	2015-10-24	• changed behavior of the console and log time stamp: if a time stamp byte followed by no other byte has been received then the time stamp is created not before the next byte has been received
3.15	2015-11-01	• increased performance at high data rates • new default arguments in worker script function: showTextInputDialog, showMultiLineTextInputDialog, showGetItemDialog, showGetIntDialog, showGetDoubleDialog, showFileDialog,showDirectoryDialog, messageBox, showYesNoDialog • sequence script function changed: messageBox • sequence script function added: showYesNoDialog, showTextInputDialog, showMultiLineTextInputDialog, showGetItemDialog, showGetIntDialog, showGetDoubleDialog
3.16	2015-11-06	• SQL support for worker and custom log/console scripts implemented • worker script function removed: deleteObject • all created script objects (except ScriptTreeWidgetItem) are automatically deleted by the garbage collector now
3.17	2015-11-08	• bug fix: The script SQL classes did not handle byte arrays correctly.
3.18	2015-11-14	• sequence and custom console/log scripts are running in their own thread now • worker, sequence and custom console/log script function added: setBlockTime • blocked worker scripts are terminated now (after the block-time (can be set with setBlockTime) has elapsed) • custom console/log script functions changed: readFile, readBinaryFile • custom console/log script functions added: getFileSize, checkFileExists, checkDirectoryExists, createDirectory, renameDirectory, renameFile, deleteFile, deleteDirectory, deleteDirectoryRecursively, readDirectory, writeFile, writeBinaryFile
3.19	2015-11-18	• new send window menus: create script, add script • 'edit custom console script' button added (console tab) • 'edit custom log script' button added (log tab)
3.20	2015-11-21	• bug fix: possible crash while starting a worker script with a syntax error
3.21	2015-12-02	• new send area in the main window • new option in the settings dialog: target endianess • new decimal types for the decimal console and the log (uint8, uint16, uint32, int8, int16 and int32) • new data formats in the send window (binary, uint8, uint16, uint32, int8, int16 and int32)
3.22	2015-12-04	• worker script function added: getScriptFolder • Mac OS X support
3.23	2015-12-13	• bug fix: sometimes up to 3 extra bytes were written into the log if the 'write decimal into log' option was disabled and then enabled again • worker and sequence script function added: showColorDialog
3.24	2015-12-21	• the sizes of the tool box pages in the main window (Sequences, Scripts, …) are saved now

		• send history added in the main window • main window send area: the data can be sent with 'alt+enter' now • main config file lock implemented
3.24.1 (Mac OS X only)	2015-12-23	• bug fix: error while starting the external script editor on Mac OS X • QtCreator is now used for editing worker script user interfaces on Mac OS X
3.25	2015-12-28	• XML support for worker and custom console/log scripts implemented • new color dialog for selecting the colors of the consoles (background, receive data …) • new manual chapter: 'Configuration files'
3.26	2015-12-30	• bug fix: ◦ possible crash while saving the content of a script plot widget/window ◦ menu 'Config/create new config' did not work correctly • settings dialog elements adjusted (especially for Linux and Mac OS X)
3.27	2016-01-04	• send history: create script button added • ScriptTcpClient, ScriptUdpSocket and ScriptSerialPort functions added: canReadLine, readLine, readAllLines • ScriptPlotWidget and ScriptPlotWindow: ◦ the x-range, y-min and y-max input fields accept floats now ◦ function setMaxDataPointsPerGraph added ◦ QCustomPlot has been updated to version 1.3.2 ◦ show legend check box added ◦ showHelperElements has new arguments ◦ signal plotMousePressSignal added • ScriptThread functions and signals added/: setScriptState, getScriptTableName, globalStringChangedSignal, globalDataArrayChangedSignal, globalUnsignedChangedSignal, globalSignedChangedSignal, setGlobalRealNumber, getGlobalRealNumber, globalRealChangedSignal
3.28	2016-01-10	• a simple default script editor is included now • the sequences- and worker-scripts in the main window tool box are organized in a list view now • ScriptPlotWidget and ScriptPlotWindow: ◦ show legend check box added ◦ showHelperElements has new arguments ◦ signal plotMousePressSignal added • ScriptThread function added: getScriptTableName
3.29	2016-01-16	• bug fix: crash if the start button (main window scripts tab) has been pressed with an empty script list • GUI's of paused worker-scripts are disabled (greyed out) now • ScriptThread functions added: currentCpuArchitecture, productType, productVersion • new sequences and scripts are inserted at the front of their tables now • new menus in the script and the send window: move up/down • ScriptPlotWidget and ScriptPlotWindow new function added: setUpdateInterval • script editor: ◦ the line number is displayed now ◦ zoom in/out implemented
4.00	2016-01-30	• new features: ◦ script debugger (worker-scripts) ◦ custom script widget • new script widget class: ScriptCanvas2D • ScriptTableWidget functions changed/added: insertWidget, getWidget, cellSelectionChangedSignal, getAllSelectedCells,, addCanvas2DWidget • ScriptWidget functions added: getClassName, height
4.01	2016-02-05	• main window send area: sequence-scripts can be used now • main window 'Top' button added • sequence and custom console/log scripts: debug support added • new custom script widget: ScriptWebView

		• ScriptCanvas2D functions added: print, saveToFile • new command-line argument: -P'plug-in path'
4.02	2016-02-07	• ScriptThread function added: availableSerialPorts • script editor: multi-document support, 'open all included scripts' button added • ScriptSerialPort, ScriptUdpSocket and ScriptTcpClient function added: writeString
4.03	2016-02-13	• new script editor buttons and menus: new script, edit user interface • script editor auto completion and call tips for several objects/functions added: ◦ scriptThread object ◦ several QtScript functions and objects • script window and send window menu: 'edit all scripts' • ScriptThread functions added: createProcessAsynchronous, waitForFinishedProcess, getProcessExitCode, killProcess, terminateProcess, writeToProcessStdin, readAllStandardOutputFromProcess, readAllStandardErrorFromProcess, stringToArray, addStringToArray • sequence and custom console/log script functions added: byteArrayToString, byteArrayToHexString, stringToArray, addStringToArray • new example script: TestProcess
4.04	2016-02-23	• ScriptThread functions added: getScriptCommunicatorFolder, getScriptArguments, zipDirectory, zipFiles, extractZipFile and getUserDocumentsFolder • new default argument in ScriptThread.ceateProcessAsynchronous • new command-line argument: -minScVersion'version', -A'argument' and -L'library path' • new main window menu: check for updates • new file types: .sce (ScriptCommuncator executable) and .scez (ScriptCommuncator executable, zipped) • new dialog: create sce file
4.05	2016-02-27	• bug fix: deleting the scez temporary folder failed in some cases • drag&drop for several GUI elements implemented
4.06	2016-03-05	• bug fixes: wrong 'send history button' text after sending the history, the console message color has not been saved • new log option: 'append time stamp at log file name' • new main window button: reopen (is only visible if the log option 'append time stamp at log file name' is selected) • the default text-log file extension is 'txt' now
4.07	2016-03-12	• new ScriptWidget function: setPaletteColorRgb • new script widget: ScriptFontComboBox • new and changed ScriptTextEdit functions: lockScrolling, insertHtml and insertPlainText • new and changed ScriptThread functions: sendDataArray, sendString, sendCanMessage and addTabsToMainWindow
4.08	2016-03-19	• bug fix: 0xd ('\r') was changed to 0xa ('\n') in the data input fields (main and send window) if the format was changed to 'ascii' • file paths in configuration files are store relative to the configuration file in which they occur if they have the same root path (on windows for example C:\)
4.09		• Linux: changed the config-file directory to '/${USER}/home/.config/ScriptCommunicator' • send history: repetition count added • main window: quit button added • new ScriptThread function: addToolBoxPagesToMainWindow

GUI documentation

Main window



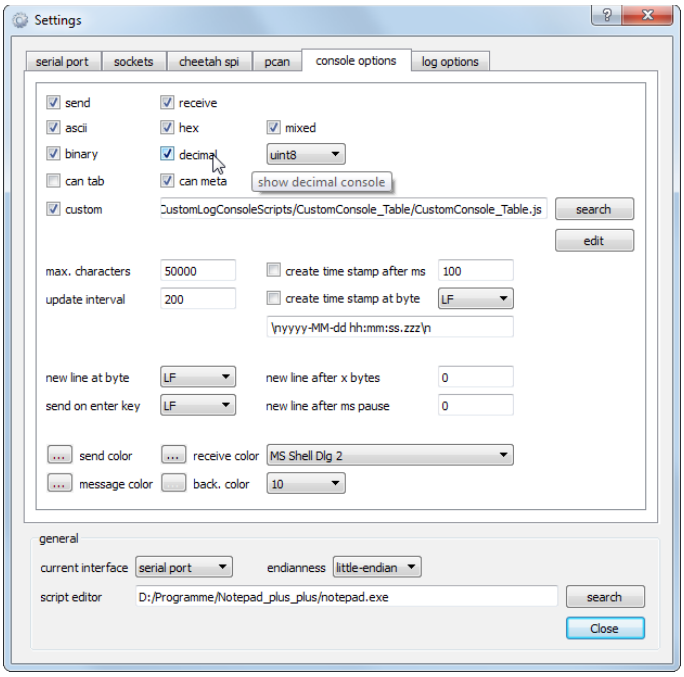
The main window contains:

- the consoles which shows:
  - the sent data
  - the received data
  - time stamps
  - messages added with the add message dialog
- a multi-line send area
- the send input check box: if checked all text entered in a console will be sent
- a menu and the following buttons:
  - Connect: connects the main interface (the configuration of the main interface is done in the configure dialog)
  - Disconnect: disconnects the main interface
  - Settings: shows the settings dialog
  - Sending: shows the send dialog
  - Scripts: shows the scripts dialog
  - Clear button: clears all consoles
  - Lock scrolling: prevents the automatic scrolling in the consoles (if new data is added to a console the cursor moves to the end of the console)
  - Add message: shows the add message dialog
  - Top: brings all windows to top/foreground
  - Reopen: reopen all logs (is only visible if the log option 'append time stamp at log file name' is selected)
- if the current connection type is serial port:
  - a check box for setting/clearing the DTR (data terminal ready) pin (default=1)
  - a check box for setting/clearing the RTS (request to send) pin (default=0)

- a sequence area in which the sequences from the send window can be sent
- a script area in which the scripts from the script window can be started (normal and in a script-debugger, stopped and paused
- a find text in console area in which a console text can be searched (in the visible console)
- the send history

Note: If the 'check for updates' menu is clicked ScriptCommunicator tries to read https://scriptcommunicator.codeplex.com/documentation. To connect to this address ScriptCommunicator uses the proxy settings from the sockets tab in the settings windows.

Settings dialog

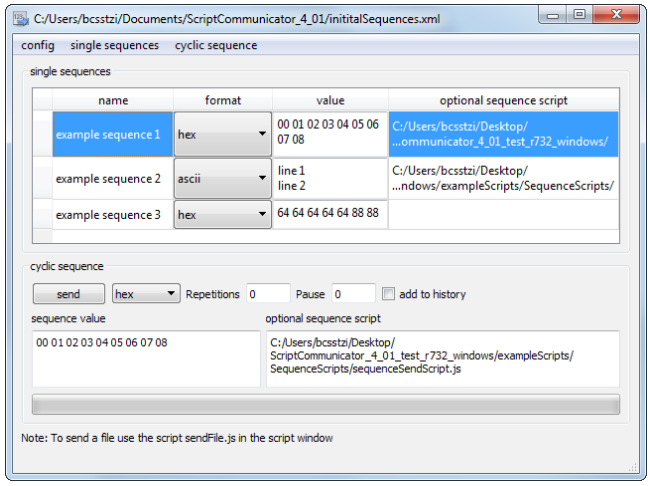


In the settings dialog following settings can be accessed:

- the main interface settings (serial port tab, sockets tab, cheetah SPI tab and PCAN tab)
- the console settings (the console options tab)
- the log settings (the log options tab)
- the path to the external script editor (for editing a script in the script window)

Note:
The script interface for the custom console and the custom log is explained in chapter Custom console/log scripts.

Send dialog



The send dialog contains the single sequence table and a cyclic sequence area.

Single sequence table

In the sequence table send sequences can be stored. The right mouse button is used to send a sequence.

Note:
A selected row in the sequence table can be moved up or down while holding the left mouse button at the row and moving the mouse up and/or down.

Cyclic sequence area

The cyclic send area contains following:

- a data input field
- a script path filed
- a send button which sends the data
- a format combo box which sets the format of the data which shall be sent
- a Repetition field which sets the number repetitions (if 0 is entered the data is sent only once, if 1 is entered the data is sent twice)
- a pause field which sets the pause in milliseconds between two repetitions
- a progress bar which shows the progress while sending data

In addition to the simple sending of a sequence a sequence script can be used (see chapter Sequence script).

Note: If ScriptCommunicator is connected to a PCAN interface the send bytes have the following meaning:

- Byte 0= message type (0=standard, 1=standard remote-transfer-request, 2=extended, 3= extended remote-transfer-request)
- Byte 1-4= can id
- Byte 5-12= the data.

If more then 8 data bytes are given, several can messages with the same CAN id will be sent.

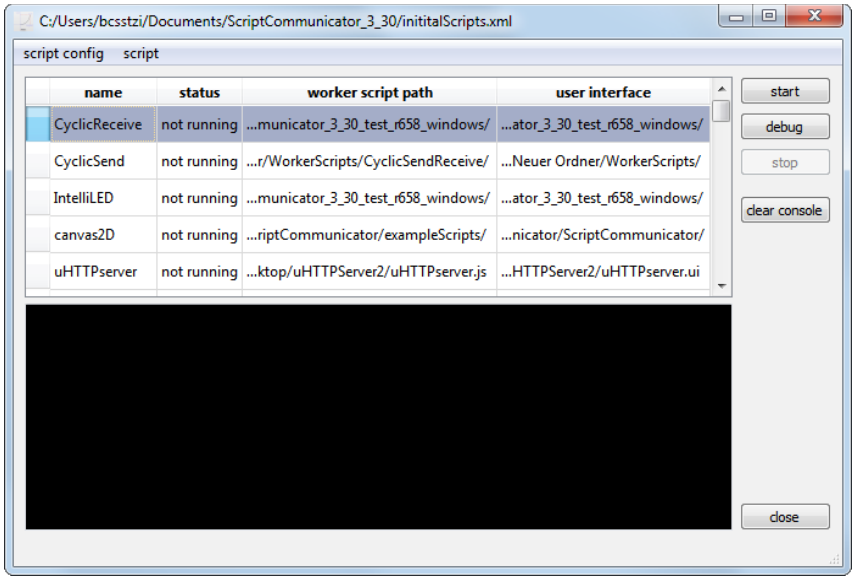
Change the sequence configuration

Following is implemented:

- create a new sequence: Ctrl+N or menu single sequence/new sequence
- add a sequence script: Ctrl+A or menu single sequence/add sequence script
- create a sequence script: Ctrl+C or menu single sequence/add sequence script
- edit a sequence script: Ctrl+E or menu single sequence/edit sequence script
- remove a sequence script: menu single sequence/remove sequence script
- delete a sequence: select the corresponding sequence and press Ctrl+D or menu single sequence/delete sequence
- move the selected sequence up: select the corresponding sequence and press Ctrl+Up or menu sequence/move up
- move the selected sequence down: select the corresponding sequence and press Ctrl+Down or menu sequence/move down
- edit all sequence scripts: press Ctrl+Shift+A or menu single sequence/edit all sequence scripts
- save the sequence config: Ctrl+S or menu config/save config
- save the sequence config under a new name (save as): Menu config/save config as
- load another sequence config: Ctrl+L or menu config/load config
- unload the current sequence config: menu config/unload config

Scripts dialog

In this window worker scripts (see chapter Worker scripts) can be added and executed.



This dialog contains following functionality:

- Change the script configuration (the content of the script table)
- Create and edit scripts and ui (user interface) files
- Load and save the script configuration (the content of the script table)
- Start, debug, pause and stop scripts

Note:
A selected row in the script table can be moved up or down while holding the left mouse button at the row and moving the mouse up and/or down.

Change the script configuration

Following is implemented:

- Add scripts to the script table: Ctrl+A or menu script/addscript
- Remove script from the script table: select the corresponding row and press Ctrl+R or menu script/remove script
- Change the path to a script in the script table: double click on the corresponding entry in path column
- Add an ui file to a script: double click on the corresponding empty entry in ui column
- Remove an ui file from the script table: select the corresponding row and press Ctrl+Shift+R or menu script/remove ui
- Change the path to an ui file in the script table: double click on the corresponding entry in ui column
- move the selected script up: select the corresponding script and press Ctrl+Up or menu script/move up
- move the selected script down: select the corresponding script and press Ctrl+Down or menu script/move down

Note:
If a script is added and in the same directory a file with the name “scriptName”.ui exists then it will be added automatically to the ui column. Example: If the script name is myScript.js then the file myScript.ui will be added automatically.

Create and edit scripts/ui (user interface) files

Following is implemented:

- Edit a script (with an external script editor): select the corresponding row and press Ctrl+E menu script/edit script
- Create scripts (from template): Ctrl+C or menu script/create script (after pressing this menu 2 dialogs appear (first the template file must be chosen and then the name/path of the new script))
- Edit an ui file (with QtDesigner): select the corresponding row and press Ctrl+Shift+E or the menu script/edit ui
- Create ui file: select the corresponding row and press Ctrl+Shift+C or the menu script/create ui
- Edit all scripts: press Ctrl+Shift+A or menu script/edit all sequence scripts

Load and save the script configuration (the content of the script table)

Following is implemented:

- Load a script configuration from a file: Ctrl+L or menu script config/load config
- Unload the current script configuration: Ctrl+U or menu script config/unload config
- Save the current script configuration: Ctrl+S or menu script config/save config

- Save config as (opens a save dialog, where a file name can be chosen): Ctrl+Shift+S or menu script config/save config as

Start, debug, pause and stop scripts

Following is implemented:

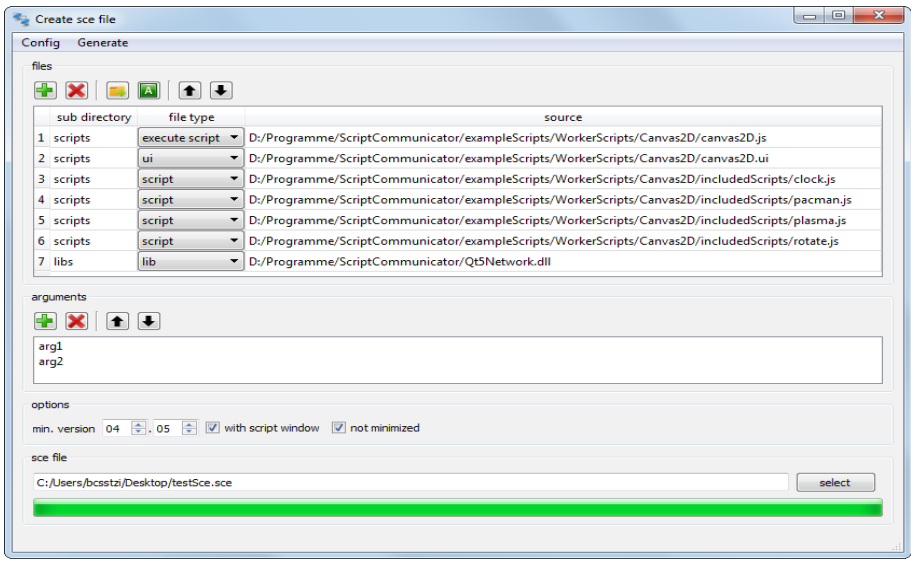
- Start a script: select the corresponding row and press the start button
- Debug a script: select the corresponding row and press the debug button (the script will be executed with an attached script debugger)
- Pause a running script: select the corresponding row and press the pause button
- Stop a running script: select the corresponding row and press the stop button

Note:
If ScriptCommunicator is closed it saves the state (running, paused and not running) of all scripts. If ScriptCommunicator is started again all script status are restored.
This means if a script is running while ScriptCommunicator is closing then this script will be automatically started after ScriptCommunicator has been started the next time.

Create sce file dialog

In this window sce (ScriptCommunicator executable) and scez (ScriptCommunicator executable, zipped) files can be created (see chapter SCE files and SCEZ files). To open this dialog go to the script window and click the 'script/create sce file' menu.

Note: ScriptCommunicator puts all necessary files to executes one ore more scripts in these files/folders. If ScriptCommunicator is started with this kind of file then all scripts marked with 'executable script' are started (see chapter Command-line mode).



Load and save the sce file configuration (the content of the complete window)

Following is implemented:

- Load a configuration from a file: Ctrl+L or menu Config/load config

- Unload the current configuration: Ctrl+U or menu Config/unload config
- Save the current configuration: Ctrl+S or menu Config/save config
- Save config as (opens a save dialog, where a file name can be chosen): Ctrl+Shift+S or menu Config/save config as

Generate a sce or scez file

To create a sce or a scez file the corresponding entries in the 'Generate' menu must be used.

Note: To generate a sce or a scez file at least one 'executable script' must be in the file table (a file marked with this type is started by ScriptCommunicator as script) and the sce file name must be set.

File table

The file table contains following columns:

- sub directory: contains the directory of the file entry inside the sce folder
- file type: the type of the file entry
- source: the source of the file entry

The file type is used by ScriptCommunicator to suggest the sub directory (which can be changed manually).
Furthermore following file types have a special meaning:

- executable script: a file marked with this type is started by ScriptCommunicator (during the sce or scez file execution)the
- lib: the sub directory of this file is added to the ScriptCommunicator library path (if you need extra libraries (e.g. for a custom widget) then add the libraries with this file type)
- plugin: the sub directory of this file is added to the ScriptCommunicator plug-in path (if you need extra plug-ins (e.g. for a custom widget) then add the plug-ins with this file type)

Arguments list

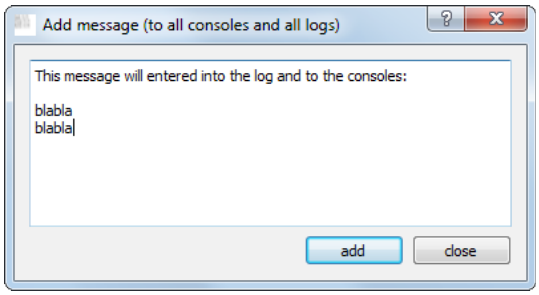
All arguments which are added here can be read by the worker scripts with QStringList getScriptArguments(void).

Options and sce file area

In this area following options can be set:

- min. version: the minimum version of ScriptCommunicator which is needed to execute the current scripts
- with script window: the script windows shall be shown (per default minimized)
- not minimized: the script window shall not be minimized
- sce file: the sce file path

Add message dialog



In this dialog a message can be entered which will be shown in the consoles and be written into the logs.

Sending and receiving a file

To send a file the script sendFile.js (exampleScripts/WorkerScripts/SendFile) can be used.
To receive a file the script receiveFile.js (exampleScripts/WorkerScripts/ReceiveFile) can be used

Configuration files

ScriptCommunicator has 4 configuration files:

- the main configuration file
- the sequences configuration file
- the scripts configuration file
- the sce configuration file

Note: File paths in configuration files are store relative to the configuration file in which they occur if they have the same root path (on windows for example C:\).

Main configuration file

The main configuration file contains:

- all settings from the settings dialog
- the sizes and positions of all ScriptCommunicator windows
- the data of the cyclic send area in the send window
- the send history
- the data of the send area in the main window
- the name/path of the used sequence configuration file
- the name/path of the use script configuration file

To create/use a new main configuration file the 'Config' menu in the main window can be used (see chapter Main window).

Sequence configuration file

The sequence configuration file contains all entries of the sequence table in the send window. To load, unload or save the sequence configuration file the 'config' menu can be used (see chapter Change the sequence configuration).

Script configuration file

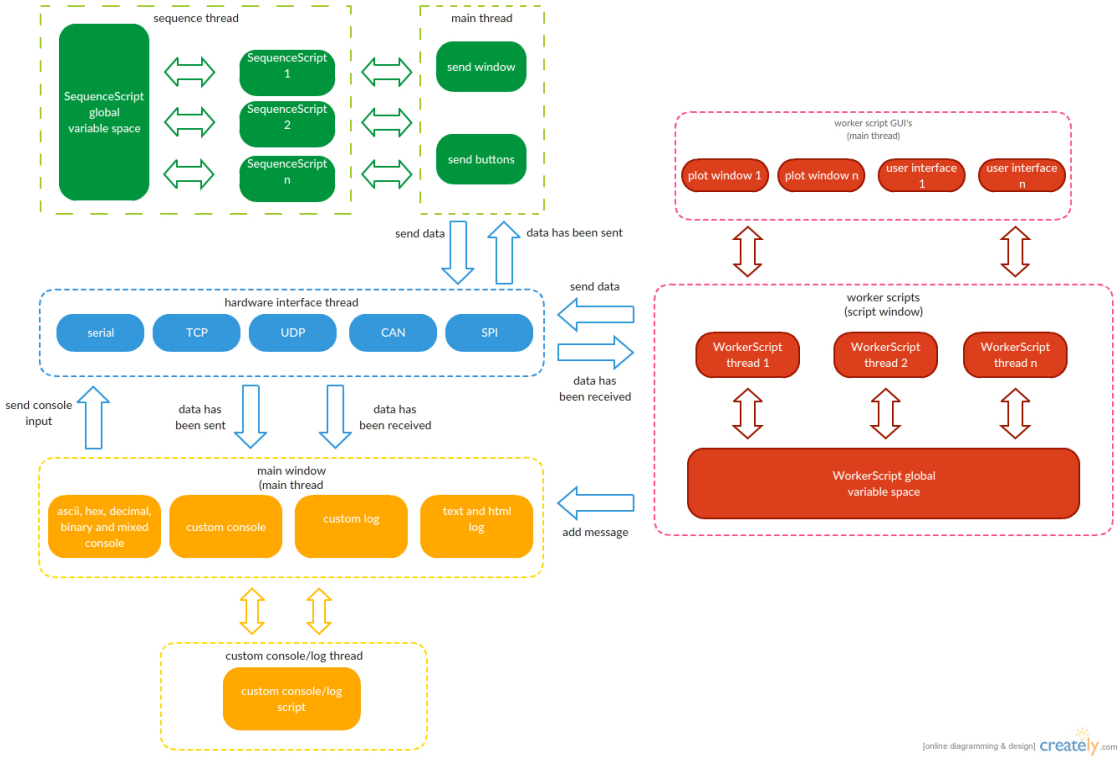
The script configuration file contains all entries of the script table in the script window. To load, unload or save the script configuration file the 'script config' menu can be used (see chapter Change the script configuration).

SCE configuration file

The sce configuration file contains the content of the complete create sce file dialog. To load, unload or save the sce configuration file the 'Config' menu can be used (see chapter Load and save the sce file configuration (the content of the complete window)).

Internal architecture

The following picture illustrates the internal architecture of ScriptCommunicator.



Script interface

The script interface of ScriptCommunicator is able the execute QtScript (similar to JavaScript) files. This is done with the Qt class QScriptEngine. All standard QtScript functionalities are included. See:

http://qt-project.org/doc/qt-5.0/qtscript/ecmascript.html#value-properties

http://www.trinitydesktop.org/docs/qt4/scripting.html

Since QtScript and JavaScript are base on ECMAScript a good JavaScript book can also be used as language reference (the core features).

ScriptCommunicator provides 3 different script types:

- worker scripts (chapter Worker scripts)
- sequence scripts (chapter Sequence script)
- custom console/log scripts (chapter Custom console/log scripts)

Script debugging

To debug scripts ScriptCommunicator uses the Qt script debugger. For more information see http://doc.qt.io/qt-4.8/qtscriptdebugger-manual.html.

Note: If a scripts runs in the debugger then it is executed in the main thread (and not in is own thread).

Worker scripts

Worker scripts can be added in the script window. In this scripts complex functions can be implemented (sending/receiving data, file operations …).

Every worker script runs in his own thread, therefore ScriptCommunicator can not be blocked by a worker script directly. The GUI of a worker script runs in the main thread (a call to a script GUI element normally calls a function in the main thread), therefore to many calls to script GUI elements can block ScriptCommunicator.

Worker scripts are QtScript scripts (see chapter Script interface). The worker script interface extends the standard QtScript functionality. These extended functionality is described in the following chapters.
Note:
A worker script can have a user interface (created with QtDesigner (is included) or QtCreator).
To load a user interface following can be done:

- set the path to the user interface in the script window (the user interface will be loaded automatically when the script starts)
- or call scriptThread.loadUserInterfaceFile() in the worker script (to load a user interface manually)

The supported GUI elements and their classes are described in chapter User interface classes

Note:

In the main function (all code outside a function) only the script initialization code should be placed. The working code should be placed in asynchronous function calls (like timer callbacks or data receive callbacks). If the main function has been left the script does not stop. To stop a script call scriptThread.stopScript() or press the stop button (main or script window).
Example:

```
//Asynchronous data receive callback.
function dataReceived()
{
	//Working code.
	if(workDone)
	{
		scriptThread.stopScript();
	}
}
//Initialization code.
var workDone = false;
scriptThread.appendTextToConsole('script has started');
scriptThread.dataReceivedSignal.connect(dataReceived);
```

If the worker code shall be placed in the main function then scriptThread.scriptShallExit() should be called to check if the script must exit.
Example:

```
scriptThread.appendTextToConsole('script has started');
while(!scriptThread.scriptShallExit())
{
	//Working code.
}
```

Command-line mode

ScriptCommunicator has a command-line mode. This means that worker scripts which shall be executed can be given as program arguments.
Example:

```
ScriptCommunicator.exe -withScriptWindow -notMinimized -minScVersion04.04 -PC:/dir1
-LC:/dir2 -Aarg1 -Aarg2 C:/script1.js C:/script2.js
```

possible command-line arguments:

- -withScriptWindow: the script windows shall be shown (per default minimized)
- -notMinimized: the script window shall not be minimized
- -P: adds an additional folder to the ScriptCommunicator plug-in (custom script widgets) search path
- -minScVersion: the minimum version of ScriptCommunicator which is needed to execute the current scripts
- -A: a script command-line argument (these arguments can be read by worker scripts with the function QStringList getScriptArguments(void)
- -L: adds an additional library path (e.g. for loading a custom script widget)
- scripts, sce files (see chapter SCE files) or scez files (see chapter SCEZ files) separated by a space

Command-line mode limitations:

- no ScriptCommunicator window is visible per default (only the script window can be shown (-withScriptWindow))

Program termination without the argument withScriptWindow:

- if all worker scripts are stopped ScriptCommunicator exits automatically
- a call to scriptThread.exitScriptCommunicator() exits ScriptCommunicator

Program termination with the argument withScriptWindow:

- ScriptCommunicator exits automatically if the script window has been closed
- a call to scriptThread.exitScriptCommunicator() exits ScriptCommunicator

SCE files

The sce file (ScriptCommunicator executable) is a possibility to put all command-line arguments in one file. This file is a simple XML file and has following structure:

```
<ExecutableConfig version="04.04"> <!--ScriptCommunicator version with which this
file has been created-->
  <Scripts>
    <Script path="./scripts/scrip1.js"/><!--script which shall be started-->
    <Script path="./scripts/scrip2.js"/><!--script which shall be started-->
  </Scripts>
  <LibraryPaths>
    <LibraryPath path="./libs1"/><!--additional library path-->
    <LibraryPath path="./libs2"/><!--additional library path-->
  </LibraryPaths>
  <PluginPaths>
    <PluginPath path="./plugins1"/><!--additional plugin path-->
    <PluginPath path="./plugins2"/><!--additional plugin path-->
  </PluginPaths>
  <ScriptArguments>
    <ScriptArgument value="arg1"/><!--script argument (getScriptArguments())-->
    <ScriptArgument value="arg2"/><!--script argument (getScriptArguments())-->
  </ScriptArguments>
  <Options withScriptWindow="1" notMinimized="1" minScVersion="4.4"/>
<!--these 3 attributes have the same meaning as the corresponding command-line
argument-->
</ExecutableConfig>
```

For creating a sce file the Create sce file dialog shall be used (but it can created manually too).

SCEZ files

A scez file (ScriptCommunicator executable, zipped) is a zipped sce file (including all necessary files and folders). To ensure the data integrity of a scez file a SHA-512 hash is appended.

Note: To create a scez file the Create sce file dialog must be used.

If ScriptCommunicator is started with this kind of file then ScriptCommunicator:

- checks the SHA-512 hash
- unzips the scez file into the temporary folder 'user documents folder'/'time in milliseconds since epoche'
- starts all executable scripts
- deletes the temporary folder on exit

void stopScript(void)

This script function can be added to a worker script. It is called if the script shall be informed that it will be stopped (for example if the user presses the stop button)

Example:

```
function stopScript()
{
    scriptThread.appendTextToConsole("script has been stopped ");
}
```

The scriptThread object/class

The scriptThread object is the main interface object for accessing ScriptCommunicator functions from script. The functions and signals which can be used from script are described in the following chapters.

Main interface

The following functions can be used to send and receive data with the main interface.

Note:
The main interface is the interface which can be adjusted in the settings window (worker scripts can create/use own interfaces too (see chapter Separate interfaces)).

bool sendDataArray(QVector<unsigned char> data, int repetitionCount=0, int pause=0, bool addToMainWindowSendHistory=false)

Sends a data array with the main interface.

Arguments:

- data: The data array
- repetitionCount: The data array is repeated until the number has been reached
- pause: The pause (milliseconds) between two repetitions
- addToMainWindowSendHistory: True if the data shall be added to the send history in the main window

Return: True on success

Example:

```
var array = Array(1,2,3,4);
var result = scriptThread.sendDataArray(array);
```

bool sendString(QString string, int repetitionCount=0, int pause=0, bool addToMainWindowSendHistory=false)

Sends a string with the main interface.

Arguments:

- string: The string
- repetitionCount: The data array is repeated until the number has been reached
- pause: The pause (ms) between two repetitions
- addToMainWindowSendHistory: True if the data shall be added to the send history in the main window

Return: True on success

Example:

```
var result = scriptThread.sendString("test string");
```

bool sendCanMessage(quint8 type, quint32 canId, QVector<unsigned char> data, int repetitionCount=0, int pause=0, bool addToMainWindowSendHistory=false)

Sends a can message with the main interface (in MainInterfaceThread). If more then 8 data bytes are given several can messages with the same can id will be sent.

Arguments:

- type: The can message type (0=standard, 1=standard remote-transfer-request, 2=extended, 3= extended remote-transfer-request)
- canId: The can id
- data: the can data
- repetitionCount: The data message is repeated until the number has been reached
- pause: The pause (ms) between two repetitions
- addToMainWindowSendHistory: True if the data shall be added to the send history in the main window

Return: True on success

Example:

```
var result = scriptThread.sendCanMessage(2, 0x0f, Array(0,0,0,0));
```

bool isConnectedWithCan(void)

Returns true if the main interface is a CAN interface (and is connected).

Example:

```
if(scriptThread.isConnectedWithCan())
{
	var result = scriptThread.sendCanMessage(2, 0x0f, Array(0,0,0,0), 0, 0);
}
```

bool isConnected(void)

Returns true if the main interface is connected.

Example:

```
if(scriptThread.isConnected())
{
	scriptThread.sendString("test string");
}
```

void disconnect(void)

Disconnects the main interface.

bool connectPcan(quint8 channel, quint32 baudrate, quint32 connectTimeout = 2000, bool busOffAutoReset = true, bool powerSupply = false, bool filterExtended = true, quint32 filterFrom = 0, quint32 filterTo = 0x1fffffff)

Connects the main interface (PCAN).

Note: A successful call will modify the corresponding settings in the settings dialog.

Arguments:

- channel: The PCAN channel
- baudrate(kBaud): The baudrate. Possible values are:
  1000, 800, 500, 250, 125,100,95,83,50,47,33,20,10,5.
- connectTimeout: Connect timeout (ms)

- busOffAutoReset: True if the PCAN driver shall reset automatically the CAN controller of a PCAN Channel if a bus-off state is detected
- powerSupply: True if the external 5V on the D-Sub connector shall be switched on
- filterExtended: True if the filer message type is extended (29-bit identifier) or false if the filter message type is standard (11-bit identifier)
- filterFrom: The lowest CAN ID to be received
- filterTo: The highest CAN ID to be received

Return: True on success

Example:

```
if(scriptThread.connectPcan(1, 1000))
{
	scriptThread.appendTextToConsole ("connectPcan succeeded");
}
```

bool connectSocket(bool isTcp, bool isServer, QString ip, quint32 partnerPort, quint32 ownPort, quint32 connectTimeout = 5000)

Connects the main interface (UDP or TCP socket).

Note: A successful call will modify the corresponding settings in the settings dialog.

Arguments:

- isTcp: True for TCP and false for UDP
- isServer: True if the connection type is a (TCP) server
- ip: The partner ip address.
- partnerPort: The partner port
- ownPort: The own port
- connectTimeout: Connection timeout (ms)

Return: True on success

Example:

```
if(scriptThread.connectSocket(false, false, "127.0.0.1", 111, 112))
{
	scriptThread.appendTextToConsole ("connectSocket succeeded (UDP socket)");
}
```

bool connectSerialPort(QString name, qint32 baudRate = 115200, quint32 connectTimeout= 1000, quint32 dataBits = 8, QString parity = "None", QString stopBits = "1", QString flowControl = "None")

Connects the main interface (serial port).

Note: A successful call will modify the corresponding settings in the settings dialog.

Arguments:

- name: True serial port name
- baudrate: The baudrate
- connectTimeout: Connection timeout (ms)
- dataBits: The number of data bits
- parity: The parity. Possible values are: "None ", "Even ", "Odd ", "Space" and "Mark"
- stopBits: The number of stop bits. Possible values are: "1 ", "1.5" and "2"
- flowControl: The flow control. Possible values are: "RTS/CTS", "XON/XOFF" and "None"

Return: True on success

Example:

```
if(scriptThread.connectSerialPort("COM1"))
```

```
{
	scriptThread.appendTextToConsole ("connectSerialPort succeeded");
}
```

void setSerialPortPins(bool setRTS, bool setDTR)

Sets the serial port (main interface) RTS (request to send) and DTR (data terminal ready) pins.
Note: A call to this function changes the value of the RTS (request to send) and the DTR (data terminal ready) check box in the main window.
Arguments:

- setRTS: true=set the pin to 1, false=set the pin to 0
- setDTR: true=set the pin to 1, false=set the pin to 0

bool connectCheetahSpi(quint32 port, qint16 mode, quint32 baudrate, quint8 chipSelectBits = 1, quint32 connectTimeout = 1000)

Connects the main interface (cheetah spi).
Note: A successful call will modify the corresponding settings in the settings dialog.
Arguments:

- port: The cheetah spi port
- mode: The spi mode (0-3)
- baudrate: The baudrate of the interface (kHz)
- chipSelectBits: The chip select bits (1-7)
- connectTimeout: Connect timeout(ms)

Return: True on success
Example:

```
if(scriptThread.connectCheetahSpi(0, 1, 1000, 1))
{
	scriptThread.appendTextToConsole (" connectCheetahSpi succeeded");
}
```

void dataReceivedSignal(QVector<unsigned char> data)

This signal is emitted if data has been received with the main interface (only if the main interface is not a CAN interface, use canMessagesReceivedSignal if the main interface is a CAN interface). Scripts can connect a function to this signal.
Arguments:

- data: The received data

Example:

```
function dataReceivedSlot(data)
{
	scriptThread.appendTextToConsole("data received: " + data);
}

//connect the dataReceivedSlot function to the dataReceivedSignal signal
scriptThread.dataReceivedSignal.connect(dataReceivedSlot)
```

void canMessagesReceivedSignal(QVector<quint8> types, QVector<quint32> messageIds, QVector<quint32> timestamps, QVector<QVector<unsigned char>> data)

This signal is emitted if a can message (or several) has been received with the main interface. Scripts can connect a function to this signal.
Arguments:

- types: The can types of the received can messages (0=standard, 1=standard remote-transfer-request, 2=extended, 3= extended remote-transfer-request)
- messageIds: The can ids of the received can messages
- timestamps: Time stamps for the received can messages (milliseconds since the first message has been received).
- data: The data of the received can messages

Example:

```
function canMessagesReceived(types, ids, timeStamps, data)
{
	for(var index = 0; index < types.length; index++)
	{
		scriptThread.appendTextToConsole("message received: " + ids[index]);
	}
}

//connect the dataReceivedSlot function to the dataReceivedSignal signal
scriptThread.canMessagesReceivedSignal.connect(canMessagesReceived)
```

Separate interfaces

The following functions can be used to create separate interfaces (to send and receive data without the main interface).

ScriptUdpSocket createUdpSocket(void)

Creates an UDP socket object (for more details about the script UDP socket see chapter Script UDP socket class).

Return: The created UDP socket object

Example:

```
var udpSocket = scriptThread.createUdpSocket();
```

ScriptTcpServer createTcpServer (void)

Creates a TCP server object (for more details about the script TCP server see chapter Script TCP server class).

Return: The created TCP server object

Example:

```
var tcpServer = scriptThread.createTcpServer();
```

ScriptTcpClient createTcpClient (void)

Creates a TCP client object (for more details about the script TCP client see chapter Script TCP client class).

Return: The created TCP client object

Example:

```
var tcpClient = scriptThread.createTcpClient();
```

ScriptCheetahSpi createCheetahSpiInterface(void)

Creates a cheetah SPI interface object (for more details about the script cheetah SPI interface see chapter Script cheetah SPI class).

Return: The created cheetah SPI interface object

Example:

```
var spi = scriptThread.createCheetahSpiInterface();
```

ScriptPcan createPcanInterface(void)

Creates a PCAN interface object (for more details about the script PCAN interface see chapter Script PCAN class).

Return: The created PCAN interface object

Example:

```
var pcan = scriptThread.createPcanInterface();
```

ScriptSerialPort createSerialPort (void)

Creates a serial port object (for more details about the script serial port see chapter Script serial port class).

Return: The created serial port object

Example:

```
var serialPort = scriptThread.createSerialPort();
```

Standard dialogs

In this chapter all available standard dialog are described.

QString showFileDialog (bool isSaveDialog, QString caption, QString dir, QString filter, QWidget parent=0)*

Shows a save file dialog (QFileDialog::getSaveFileName) or an open file dialog (QFileDialog::getOpenFileName).

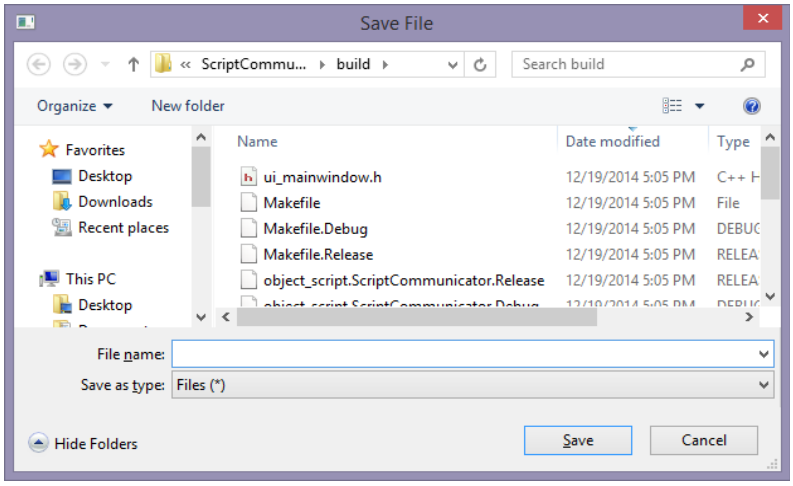
Arguments:

- isSaveDialog: True for a QFileDialog::getSaveFileName and false for a QFileDialog::getOpenFileName dialog
- caption: The caption of the dialog
- dir: The initial dir for showing the dialog
- filter: Filter for the file dialog (for more details see QFileDialog, http://doc.qt.io/qt-4.8/qfiledialog.html)
- parent: The parent of this dialog (see chapter QWidget* getWidgetPointer(void) for more details)

Return: The path of the selected file

Example:

```
var path = scriptThread.showFileDialog(true, "Save File", "c:/TestDir/","Files (*)")
```



QString showDirectoryDialog(QString caption, QString dir, QWidget parent=0)*

Shows a QFileDialog::getExistingDirectory dialog (is used to select a directory).

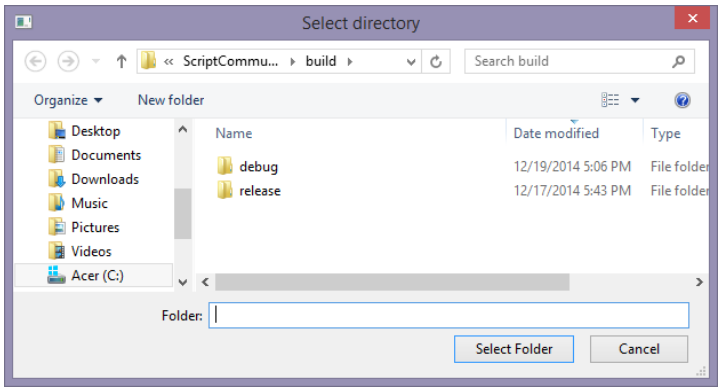
Arguments:

- caption: The caption of the dialog
- dir: The initial dir for showing the dialog
- parent: The parent of this dialog (see chapter QWidget* getWidgetPointer(void) for more details)

Return: The path of the selected directory

Example:

```
var dir = scriptThread.showDirectoryDialog("Select directory", "c:/TestDir/");
```



QString showTextInputDialog(QString title, QString label, QString displayedText="", QWidget parent=0)*

Convenience function to get a string from the user. Shows a QInputDialog::getText dialog (line edit).

Arguments:

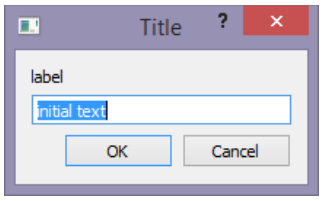
- title: The title of the dialog
- label: The label over the input area
- displayedText: The initial displayed text in the input area
- parent: The parent of this dialog (see chapter QWidget* getWidgetPointer(void) for more details)

Return: The text in the input section after closing the dialog (empty if the ok button was not pressed).

Example:

```
var input = scriptThread.showTextInputDialog("Title", "label", "initial text");
```

```
if(input != "")
{
	scriptThread.appendTextToConsole("ok button pressed: input=" + input);
}
else
{
	scriptThread.appendTextToConsole("ok button not pressed or empty input");
}
```



QString showMultiLineTextInputDialog(QString title, QString label, QString displayedText="", QWidget parent=0)*

Convenience function to get a multiline string from the user. Shows a QInputDialog::getMultiLineText dialog (plain text edit).

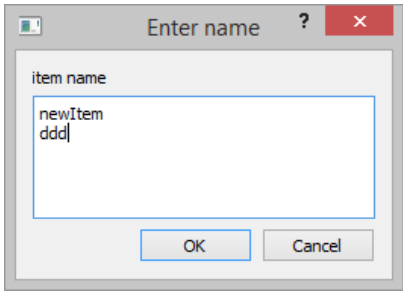
Arguments:

- title: The title of the dialog
- label: The label over the input area
- displayedText: The initial displayed text in the input area
- parent: The parent of this dialog (see chapter QWidget* getWidgetPointer(void) for more details)

Return: The text in the input section after closing the dialog (empty if the ok button was not pressed).

Example:

```
var input = scriptThread.showMultiLineTextInputDialog("Enter name", "item name",
"newItem");
if(input != "")
{
	scriptThread.appendTextToConsole("ok button pressed: input=" + input);
}
else
{
	scriptThread.appendTextToConsole("ok button not pressed or empty input");
}
```



QString showGetItemDialog(QString title, QString label, QStringList displayedItems, int currentItemIndex=0, bool editable=false, QWidget parent=0)*

Convenience function to let the user select an item from a string list. Shows a QInputDialog::getItem dialog (combobox).

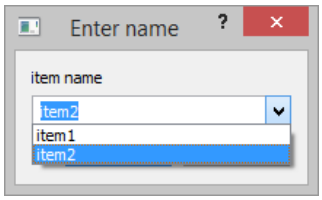
Arguments:

- title: The title of the dialog
- label: The label over the input area
- displayedItems: The displayed items
- currentItemIndex: The current combobox index
- editable: True if the combobox shall be editable
- parent: The parent of this dialog (see chapter QWidget* getWidgetPointer(void) for more details)

Return: The text of the selected item after closing the dialog (empty if the ok button was not pressed).

Example:

```
var input = scriptThread.showGetItemDialog("Enter name", "item name", Array("item1",
"item2"), 1, true);
if(input != "")
{
	scriptThread.appendTextToConsole("ok button pressed: input=" + input);
}
else
{
	scriptThread.appendTextToConsole("ok button not pressed or empty input");
}
```



QList<int> showGetIntDialog(QString title, QString label, int intialValue, int min, int max, int step, QWidget parent=0)*

Convenience function to get an integer input from the user. Shows a QInputDialog::getInt dialog (spinbox).

Arguments:

- title: The title of the dialog
- label: The label over the input area
- intialValue: The initial value.
- min: The minimum value
- max: The maximum value.
- step: The amount by which the values change as the user presses the arrow buttons to increment or decrement the value
- parent: The parent of this dialog (see chapter QWidget* getWidgetPointer(void) for more details)

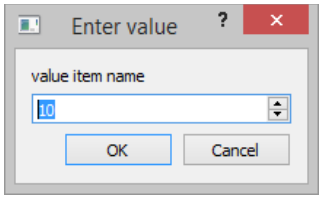
Return:

- array item 0: 1 if the ok button has been pressed, 0 otherwise
- array item 1: The value of the spinbox after closing the dialog

Example:

```
var resultArray = scriptThread.showGetIntDialog("Enter value", "value item name",
10, 0, 20, 2);
if(resultArray[0] == 1)
{
```

```
        scriptThread.appendTextToConsole("ok button pressed: input=" +
                                                    resultArray[1]);
}
else
{
        scriptThread.appendTextToConsole("ok button not pressed");
}
```



QList<double> showGetDoubleDialog(QString title, QString label, double intialValue, double min, double max, int decimals, QWidget parent=0)*

Convenience function to get a floating point number from the user. Shows a QInputDialog::getDouble dialog (spinbox).

Arguments:

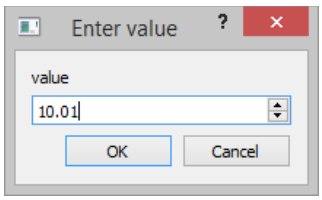
- title: The title of the dialog
- label: The label over the input area
- intialValue: The initial value.
- min: The minimum value
- max: The maximum value.
- decimals: The maximum number of decimal places the number may have
- parent: The parent of this dialog (see chapter QWidget* getWidgetPointer(void) for more details)

Return:

- array item 0: 1.0 if the ok button has been pressed, 0 otherwise
- array item 1: The value of the spinbox after closing the dialog

Example:

```
var resultArray = scriptThread.showGetDoubleDialog("Enter value", "value", 10, 0,
20, 2);
if(resultArray[0] >= 1.0)
{
        scriptThread.appendTextToConsole("ok button pressed: input=" +
                              resultArray[1]);
}
else
{
        scriptThread.appendTextToConsole("ok button not pressed");
}
```



void messageBox(QString icon, QString title, QString text, QWidget parent=0)*

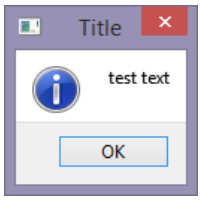
This function shows a message box.

Arguments:

- icon: The icon of the message box. Possible values are: "Information"," Warning", "Critical" and "Question"
- title: The title of the message box
- text: The text of the message box
- parent: The parent of this dialog (see chapter QWidget* getWidgetPointer(void) for more details)

Example:

```
scriptThread.messageBox("Information", "Title", "test text");
```



bool showYesNoDialog(QString icon, QString title, QString text, QWidget parent=0)*

This function shows a yes/no dialog.

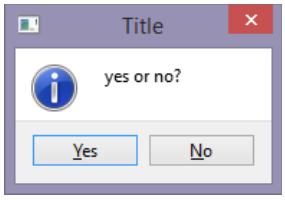
Arguments:

- icon: The icon of the dialog. Possible values are: "Information"," Warning", "Critical" and "Question"
- title: The title of the dialog
- text: The text of the dialog
- parent: The parent of this dialog (see chapter QWidget* getWidgetPointer(void) for more details)

Return: True if the user has pressed the yes button

Example:

```
if(scriptThread.showYesNoDialog("Information", "Title", "yes or no?"))
{//Yes clicked.
	//Do something.
}
```



QList<int> showColorDialog(quint8 initialRed=255, quint8 initialGreen=255, quint8 initialBlue=255, quint8 initialAlpha=255, bool alphaIsEnabled=false, QWidget parent=0)*

Convenience function to get color settings from the user.

Arguments:

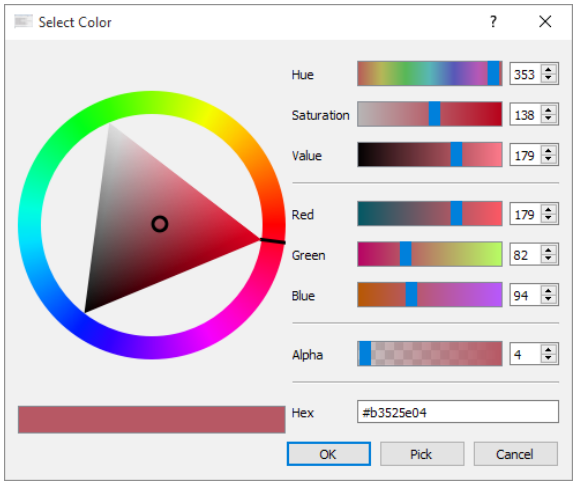
- initialRed: The initial value for red
- initialGreen: The initial value for green
- initialBlue: The initial value for blue
- initialAlpha: The initial value for alpha
- alphaIsEnabled: True if the alpha value should be visible/editable
- parent: The parent of this dialog (see chapter QWidget* getWidgetPointer(void) for more details)

Return: integer list which contains:

- element 0: 1 = OK button press, 0 OK button not pressed
- element 1: red (0-255)
- element 2: green (0-255)
- element 3: blue (0-255)
- element 4: alpha (0-255)

Example:

```
var resultArray = scriptThread.showColorDialog(1,2,3,4,true);
var data = Array();
if(resultArray[0])
{//OK clicked.
	data.push(resultArray[1]); //Red
	data.push(resultArray[2]); //Green
	data.push(resultArray[3]); //Blue
	data.push(resultArray[4]); //Alpha
}
```



Filesystem

The following function can be used to access the file system.

bool checkFileExists(QString path, bool isRelativePath=true)

Checks if a file exists.

Arguments:

- path: The file path
- isRelativePath: True if the file path is relative to the current script (which executes this function)

Return: True if the file exists and false if not

Example:

```
var result = scriptThread.checkFileExists("Testfile.txt");
```

QString createAbsolutePath(QString fileName)

Converts a relative path (relative to the current script) into an absolute path.

Arguments:

- fileName: The relative path

Return: The created absolute path

Example:

```
var absolutePath = scriptThread.createAbsolutePath("TestScript.js");
```

QString getScriptFolder(void)

Returns the folder in which the main script resides.

Example:

```
var result = scriptThread.getScriptFolder();
```

qint64 getFileSize(QString path, bool isRelativePath=true)

Returns the size of a file.

Arguments:

- path: The file path
- isRelativePath: True if the file path is relative to the current script (which executes this function)

Return: The file size if the file exists, -1 if the file doesn't exists

Example:

```
var size = scriptThread.getFileSize("Testfile.txt");
```

QString readFile (QString path, bool isRelativePath=true, quint64 startPosition=0, qint64 numberOfBytes=-1)

Reads a text file and returns the content.

Arguments:

- path: The file path
- isRelativePath: True if the file path is relative to the current script (which executes this function)
- startPosition: Start position (the file is read from this position)
- numberOfBytes: The number of bytes which shall be read. If numberOfBytes is < 0 then all bytes from startPosition are read

Return: The file as string

Example:

```
//Read the complete file.
var string = scriptThread.readFile("Testfile.txt");
//Read 20000 bytes from byte 100.
var string2 = scriptThread.readFile("Testfile2.txt", true, 100, 20000);
```

QVector<unsigned char> readBinaryFile (QString path, bool isRelativePath=true, quint64 startPosition=0, qint64 numberOfBytes=-1)

Reads a binary file and returns the content.

Arguments:

- path: The file path
- isRelativePath: True if the file path is relative to the current script (which executes this function)
- startPosition: Start position (the file is read from this position)
- numberOfBytes: The number of bytes which shall be read. If numberOfBytes is < 0 then all bytes from startPosition are read

Return: The file as byte array

Example:

```
//Read the complete file.
```

```
var array = scriptThread.readFile("Testfile.bin");
//Read 20000 bytes from byte 100.
var array2 = scriptThread.readFile("Testfile2.bin", true, 100, 20000);
```

bool writeFile(QString path, bool isRelativePath, QString content, bool replaceFile, qint64 startPosition=-1)

Writes a text file.

Arguments:

- path: The file path
- isRelativePath: True if the file path is relative to the current script (which executes this function)
- content: The content to write
- replaceFile: If replaceFile is true, the existing file will be overwritten, else the content is appended
- startPosition: If replaceFile is false then this is the start position at which the data will be written. For appending the data at the end of the file startPosition must be < 0.

Return: True on success

Example:

```
//Replace the file.
var result = scriptThread.writeFile("Testfile.txt", true, "new content", true);
//Append text.
result = scriptThread.writeFile("Testfile.txt", true, "new content", false);
//Write from position 3.
result = scriptThread.writeFile("Testfile.txt", true, "new content", false, 3);
```

bool writeBinaryFile(QString path, bool isRelativePath, QVector<unsigned char> content, bool replaceFile, qint64 startPosition=-1))

Writes a binary file file.

Arguments:

- path: The file path
- isRelativePath: True if the file path is relative to the current script (which executes this function)
- content: The content to write
- replaceFile: If replaceFile is true, the existing file is overwritten, else the content is appended
- startPosition: If replaceFile is false then this is the start position at which the data will be written. For appending the data at the end of the file startPosition must be < 0.

Return: True on success

Example:

```
var array = Array(1,2,3,4,5,6);
//Replace the file.
var result = scriptThread.writeBinaryFile("Testfile.bin", true, array, true);
//Append data.
Result = scriptThread.writeBinaryFile("Testfile.bin", true, array, false);
//Write from position 3.
result = scriptThread.writeBinaryFile("Testfile.bin", true, array, false, 3);
```

bool deleteFile(QString path, bool isRelativePath=true)

Deletes a file.

Arguments:

- path: The file path.

- isRelativePath: True if the file path is relative to the current script (which executes this function)

Return: True on success

Example:

```
var result = scriptThread.deleteFile("Testfile.txt");
```

bool renameFile(QString path, QString newName)

Renames a file.

Arguments:

- path: The file path.
- newName: The new name

Return: True on success

Example:

```
var result = scriptThread.renameFile("C:/Dir1/TestFile.txt", "C:/Dir1/newName.txt");
```

QStringList readDirectory(QString directory, bool isRelativePath=true, bool recursive=true, bool returnFiles=true, bool returnDirectories=true)

Reads the content of a directory and his sub directories.

Arguments:

- directory: The directory path
- isRelativePath: True if the directory path is relative to the current script (which executes this function)
- recursive: If true the result includes the contents of all sub directories (and their sub directories)
- returnFiles: If true the result contains all found files
- returnDirectories: If true the result contains all found directories

Return: The found entries

Example:

```
var array = scriptThread.readDirectory("TestDir");
```

bool checkDirectoryExists(QString path, bool isRelativePath=true)

Checks if a directory exists.

Arguments:

- path: The directory path.
- isRelativePath: True if the directory path is relative to the current script (which executes this function)

Return: True if the directory exists and false if not

Example:

```
var result = scriptThread.checkDirectoryExists("Testdirectory");
```

bool deleteDirectory(QString directory, bool isRelativePath=true)

Deletes a directory (must be empty).

Arguments:

- directory: The directory path.
- isRelativePath: True if the directory path is relative to the current script (which executes this function)

Return: True on success
Example:
```
var result = scriptThread.deletedirectory ("Testdir");
```

bool deleteDirectoryRecursively(QString directory, bool isRelativePath=true)

Removes the directory, including all its contents.
If a file or directory cannot be removed, deleteDirectoryRecursively() keeps going and attempts to delete as many files and sub-directories as possible, then returns false.
If the directory was already removed, the method returns true (expected result already reached).
Arguments:

- directory: The directory path.
- isRelativePath: True if the directory path is relative to the current script (which executes this function)

Return: True on success
Example:
```
var result = scriptThread.deleteDirectoryRecursively ("Testdir");
```

bool createDirectory(QString path, bool isRelativePath=true)

Creates a directory.
Arguments:

- path: The directory path.
- isRelativePath: True if the directory path is relative to the current script (which executes this function)

Return: True on success
Example:
```
var result = scriptThread.createDirectory("Testdirectory");
```

bool renameDirectory(QString path, QString newName)

Renames a directory.
Arguments:

- path: The directory path.
- newName: The new name (always relative path)

Return: True on success:
Example:
```
var result = scriptThread.renameDirectory("C:/Dir1/Testdirectory",
"C:/Dir1/newName");
```

bool zipDirectory(QString fileName, QString sourceDirName, QString comment="")

Zips a directory.
Arguments:

- fileName: The zip file name.
- sourceDirName: The source directory.
- comment: The zip file comment.

Return: True on success:
Example:
```
var result = scriptThread.zipDirectory("C:/Dir1/Test.zip", "C:/Dir1/SourceDir");
```

bool zipFiles(QString fileName, QVariantList fileList, QString comment="")

Adds files to a zip file.

Arguments:

- fileName: The zip file name.
- fileList: Contains all files. An entry consists of a string pair. The first entry of this pair is the source file name (including the absolute file path) and the second is the file name inside the zip file (including the relative path) .
- comment: The zip file comment.

Return: True on success:

Example:

```
var fileList = Array();
fileList[0] = Array("C:/file1", "file1");
fileList[1] = Array("C:/file2", "dir1/file2");
fileList[2] = Array("C:/file3", "dir2/dir/file3");
var result = scriptThread.zipFiles("C:/ZipFile.zip", fileList);
```

bool extractZipFile(QString fileName, QString destinationDirectory)

Extracts a zip file.

Arguments:

- fileName: The zip file name.
- destinationDirectory: The destination directory.

Return: True on success:

Example:

```
var result = scriptThread.extractZipFile("C:/Dir1/Test.zip", "C:/Dir1/DestDir");
```

SQL support

Worker scripts can access SQL databases. The functionality is described in chapter SQL support.

XML support

Worker scripts can access XML files with the ScriptXmlReader and the ScriptXmlWriter classes. These classes are described in chapter XML support.

To create an object of this classes following functions must be used:

ScriptXmlReader createXmlReader(void)*

Creates a XML reader.

Example:

```
var reader = cust.createXmlReader();
```

ScriptXmlWriter createXmlWriter(void)*

Creates a XML writer.

Example:

```
var writer = cust.createXmlWriter();
```

CRC functions

The following functions can be used to create various CRC's.

quint8 calculateCrc8(QVector<unsigned char> data)

Calculates a CRC8.

Arguments:

- data: The data for calculating the CRC

Return: The CRC

Example:

```
var dataArray = Array(0, 1, 2, 3, 4, 5, 6, 7)
var crc =  scriptThread.calculateCrc8(dataArray);
```

Following code is used to calculate the CRC8:

```
quint8 ScriptThread::calculateCrc8(QVector<unsigned char> data)
{
    static const quint8 crc8Table[] = {
        0x00, 0x3e, 0x7c, 0x42, 0xf8, 0xc6, 0x84, 0xba, 0x95, 0xab, 0xe9, 0xd7,
        0x6d, 0x53, 0x11, 0x2f, 0x4f, 0x71, 0x33, 0x0d, 0xb7, 0x89, 0xcb, 0xf5,
        0xda, 0xe4, 0xa6, 0x98, 0x22, 0x1c, 0x5e, 0x60, 0x9e, 0xa0, 0xe2, 0xdc,
        0x66, 0x58, 0x1a, 0x24, 0x0b, 0x35, 0x77, 0x49, 0xf3, 0xcd, 0x8f, 0xb1,
        0xd1, 0xef, 0xad, 0x93, 0x29, 0x17, 0x55, 0x6b, 0x44, 0x7a, 0x38, 0x06,
        0xbc, 0x82, 0xc0, 0xfe, 0x59, 0x67, 0x25, 0x1b, 0xa1, 0x9f, 0xdd, 0xe3,
        0xcc, 0xf2, 0xb0, 0x8e, 0x34, 0x0a, 0x48, 0x76, 0x16, 0x28, 0x6a, 0x54,
        0xee, 0xd0, 0x92, 0xac, 0x83, 0xbd, 0xff, 0xc1, 0x7b, 0x45, 0x07, 0x39,
        0xc7, 0xf9, 0xbb, 0x85, 0x3f, 0x01, 0x43, 0x7d, 0x52, 0x6c, 0x2e, 0x10,
        0xaa, 0x94, 0xd6, 0xe8, 0x88, 0xb6, 0xf4, 0xca, 0x70, 0x4e, 0x0c, 0x32,
        0x1d, 0x23, 0x61, 0x5f, 0xe5, 0xdb, 0x99, 0xa7, 0xb2, 0x8c, 0xce, 0xf0,
        0x4a, 0x74, 0x36, 0x08, 0x27, 0x19, 0x5b, 0x65, 0xdf, 0xe1, 0xa3, 0x9d,
        0xfd, 0xc3, 0x81, 0xbf, 0x05, 0x3b, 0x79, 0x47, 0x68, 0x56, 0x14, 0x2a,
        0x90, 0xae, 0xec, 0xd2, 0x2c, 0x12, 0x50, 0x6e, 0xd4, 0xea, 0xa8, 0x96,
        0xb9, 0x87, 0xc5, 0xfb, 0x41, 0x7f, 0x3d, 0x03, 0x63, 0x5d, 0x1f, 0x21,
        0x9b, 0xa5, 0xe7, 0xd9, 0xf6, 0xc8, 0x8a, 0xb4, 0x0e, 0x30, 0x72, 0x4c,
        0xeb, 0xd5, 0x97, 0xa9, 0x13, 0x2d, 0x6f, 0x51, 0x7e, 0x40, 0x02, 0x3c,
        0x86, 0xb8, 0xfa, 0xc4, 0xa4, 0x9a, 0xd8, 0xe6, 0x5c, 0x62, 0x20, 0x1e,
        0x31, 0x0f, 0x4d, 0x73, 0xc9, 0xf7, 0xb5, 0x8b, 0x75, 0x4b, 0x09, 0x37,
        0x8d, 0xb3, 0xf1, 0xcf, 0xe0, 0xde, 0x9c, 0xa2, 0x18, 0x26, 0x64, 0x5a,
        0x3a, 0x04, 0x46, 0x78, 0xc2, 0xfc, 0xbe, 0x80, 0xaf, 0x91, 0xd3, 0xed,
        0x57, 0x69, 0x2b, 0x15};

    quint8 crc  = 0xff;
    for (auto val : data)
    {
        crc = crc8Table[(crc ^ val) & 0xff];
    }
    crc = ~crc;
    return crc;
}
```

quint16 calculateCrc16(QVector<unsigned char> data)

Calculates a CRC16.

Arguments:

- data: The data for calculating the CRC

Return: The CRC

Example:

```
var dataArray = Array(0, 1, 2, 3, 4, 5, 6, 7)
var crc =  scriptThread.calculateCrc16(dataArray);
```

Following code is used to calculate the CRC16:

```
quint16 ScriptThread::calculateCrc16(QVector<unsigned char> data)
{
```

```
    static const quint16 crc16Table[256] =
    {
     0x0000, 0xC0C1, 0xC181, 0x0140, 0xC301, 0x03C0, 0x0280, 0xC241,
     0xC601, 0x06C0, 0x0780, 0xC741, 0x0500, 0xC5C1, 0xC481, 0x0440,
     0xCC01, 0x0CC0, 0x0D80, 0xCD41, 0x0F00, 0xCFC1, 0xCE81, 0x0E40,
     0x0A00, 0xCAC1, 0xCB81, 0x0B40, 0xC901, 0x09C0, 0x0880, 0xC841,
     0xD801, 0x18C0, 0x1980, 0xD941, 0x1B00, 0xDBC1, 0xDA81, 0x1A40,
     0x1E00, 0xDEC1, 0xDF81, 0x1F40, 0xDD01, 0x1DC0, 0x1C80, 0xDC41,
     0x1400, 0xD4C1, 0xD581, 0x1540, 0xD701, 0x17C0, 0x1680, 0xD641,
     0xD201, 0x12C0, 0x1380, 0xD341, 0x1100, 0xD1C1, 0xD081, 0x1040,
     0xF001, 0x30C0, 0x3180, 0xF141, 0x3300, 0xF3C1, 0xF281, 0x3240,
     0x3600, 0xF6C1, 0xF781, 0x3740, 0xF501, 0x35C0, 0x3480, 0xF441,
     0x3C00, 0xFCC1, 0xFD81, 0x3D40, 0xFF01, 0x3FC0, 0x3E80, 0xFE41,
     0xFA01, 0x3AC0, 0x3B80, 0xFB41, 0x3900, 0xF9C1, 0xF881, 0x3840,
     0x2800, 0xE8C1, 0xE981, 0x2940, 0xEB01, 0x2BC0, 0x2A80, 0xEA41,
     0xEE01, 0x2EC0, 0x2F80, 0xEF41, 0x2D00, 0xEDC1, 0xEC81, 0x2C40,
     0xE401, 0x24C0, 0x2580, 0xE541, 0x2700, 0xE7C1, 0xE681, 0x2640,
     0x2200, 0xE2C1, 0xE381, 0x2340, 0xE101, 0x21C0, 0x2080, 0xE041,
     0xA001, 0x60C0, 0x6180, 0xA141, 0x6300, 0xA3C1, 0xA281, 0x6240,
     0x6600, 0xA6C1, 0xA781, 0x6740, 0xA501, 0x65C0, 0x6480, 0xA441,
     0x6C00, 0xACC1, 0xAD81, 0x6D40, 0xAF01, 0x6FC0, 0x6E80, 0xAE41,
     0xAA01, 0x6AC0, 0x6B80, 0xAB41, 0x6900, 0xA9C1, 0xA881, 0x6840,
     0x7800, 0xB8C1, 0xB981, 0x7940, 0xBB01, 0x7BC0, 0x7A80, 0xBA41,
     0xBE01, 0x7EC0, 0x7F80, 0xBF41, 0x7D00, 0xBDC1, 0xBC81, 0x7C40,
     0xB401, 0x74C0, 0x7580, 0xB541, 0x7700, 0xB7C1, 0xB681, 0x7640,
     0x7200, 0xB2C1, 0xB381, 0x7340, 0xB101, 0x71C0, 0x7080, 0xB041,
     0x5000, 0x90C1, 0x9181, 0x5140, 0x9301, 0x53C0, 0x5280, 0x9241,
     0x9601, 0x56C0, 0x5780, 0x9741, 0x5500, 0x95C1, 0x9481, 0x5440,
     0x9C01, 0x5CC0, 0x5D80, 0x9D41, 0x5F00, 0x9FC1, 0x9E81, 0x5E40,
     0x5A00, 0x9AC1, 0x9B81, 0x5B40, 0x9901, 0x59C0, 0x5880, 0x9841,
     0x8801, 0x48C0, 0x4980, 0x8941, 0x4B00, 0x8BC1, 0x8A81, 0x4A40,
     0x4E00, 0x8EC1, 0x8F81, 0x4F40, 0x8D01, 0x4DC0, 0x4C80, 0x8C41,
     0x4400, 0x84C1, 0x8581, 0x4540, 0x8701, 0x47C0, 0x4680, 0x8641,
     0x8201, 0x42C0, 0x4380, 0x8341, 0x4100, 0x81C1, 0x8081, 0x4040 };

    quint16 crc = 0xFFFF;
    for (auto val : data)
    {
        crc = (crc >> 8) ^ crc16Table[(crc ^ val) & 0xff];
    }
    crc = ~crc;
    return crc;

}
```

quint32 calculateCrc32(QVector<unsigned char> data)

Calculates a CRC32.

Arguments:

- data: The data for calculating the CRC

Return: The CRC

Example:

```
var dataArray = Array(0, 1, 2, 3, 4, 5, 6, 7)
var crc =  scriptThread.calculateCrc32(dataArray);
```

Following code is used to calculate the CRC32:

```
quint32 ScriptThread::calculateCrc32(QVector<unsigned char> data)
{
```

```
    static bool crc32TableCreated = false;
    static quint32 crc32Table[256];

    if(!crc32TableCreated)
    {
        const quint32 CRCPOLY = 0xEDB88320;
        quint32 value;
        for (quint32 i = 0; i < 256; i++)
        {
            value = i;
            for (int j = 8; j > 0; j--)
            {
                if (value & 1)
                {
                    value = (value >> 1) ^ CRCPOLY;
                }
                else
                {
                    value >>= 1;
                }
            }
            crc32Table[i] = value;
        }

        crc32TableCreated = true;
    }

    quint32 crc = 0xFFFFFFFF;
    for (auto val : data)
    {
        crc = crc32Table[(crc ^ val) & 0xFF] ^ (crc >> 8);
    }
    crc = ~crc;
    return crc;

}
```

quint64 calculateCrc64(QVector<unsigned char> data)

Calculates a CRC64.

Arguments:

- data: The data for calculating the CRC

Return: The CRC

Example:

```
var dataArray = Array(0, 1, 2, 3, 4, 5, 6, 7)
var crc =  scriptThread.calculateCrc64(dataArray);
```

Following code is used to calculate the CRC64:

```
quint64 ScriptThread::calculateCrc64(QVector<unsigned char> data)
{
    static bool crc64TableCreated = false;
    static quint64 crc64Table[256];

    if(!crc64TableCreated)
    {
        const quint64 CRCPOLY = 0x42F0E1EBA9EA3693;
```

```
        quint64 value;
        for (quint32 i = 0; i < 256; i++)
        {
            value = i;
            for (int j = 8; j > 0; j--)
            {
                if (value & 1)
                {
                    value = (value >> 1) ^ CRCPOLY;
                }
                else
                {
                    value >>= 1;
                }
            }
            crc64Table[i] = value;
        }

        crc64TableCreated = true;
    }

    quint64 crc = 0;
    for (auto val : data)
    {
        crc = crc64Table[(crc ^ val) & 0xFF] ^ (crc >> 8);
    }
    crc = ~crc;
    return crc;
}
```

Inter-WorkerScript communication

The following functions can be used to share variables between single worker scripts (these variable are stored in ScriptCommunicator global worker script maps).

Under exampleScripts\WorkerScripts\InterWorkerScriptDataExchange an example of Inter-WorkerScript communication can be found.

void setGlobalString(QString name, QString string)

Sets a string in the global string map.
Arguments:

- name: Name of the string variable
- string: The string

QString getGlobalString(QString name, bool removeValue=false)

Returns a string from the global string map.
Arguments:

- name: Name of the variable
- removeValue: True if the variable shall be removed from the string map

Return: The read string. Returns an empty string if name is not in the string map.

void globalStringChangedSignal(QString name, QString string)

This signal is emitted if a string in the global string map has been changed. Scripts can connect a function to this signal.
Arguments:

- name: Name of the variable
- string: The content of the string

void setGlobalDataArray(QString name, QVector<unsigned char> data)

Sets a data array in the global data array map.
Arguments:

- name: Name of the variable
- data: The data array

QVector<unsigned char> getGlobalDataArray(QString name, bool removeValue=false)

Returns a data array from the global data array map.
Arguments:

- name: Name of the variable
- removeValue: True if the variable shall be removed from the data array map

Return: The read data array. Returns an empty data array if name is not in the data array map.

void globalDataArrayChangedSignal(QString name, QVector<unsigned char> data)

This signal is emitted if a data vector in the global string data vector has been changed. Scripts can connect a function to this signal.
Arguments:

- name: Name of the variable
- data: The content of data array

void setGlobalUnsignedNumber(QString name, quint32 number)

Sets an unsigned number in the global unsigned number map.
Arguments:

- name: Name of the variable
- number: The number

QList<quint32> getGlobalUnsignedNumber(QString name, bool removeValue=false)

Returns an unsigned number from the global unsigned number map.
Arguments:

- name: Name of the variable
- removeValue: True if the variable shall be removed from the unsigned number map

Return: The first element in the result list is the result status (1=name found, 0=name not found). The second element is the read value.
Example:

```
//Read the stored value
var resultArray = scriptThread.getGlobalUnsignedNumber("U_Number");
if(resultArray[0] == 1)
{
	scriptThread.appendTextToConsole("unumber: " + resultArray[1]);
}
```

void globalUnsignedChangedSignal(QString name, quint32 number)

This signal is emitted if an unsigned number in the global unsigned number map has been changed. Scripts can connect a function to this signal.
Arguments:

- name: Name of the variable
- number: The value of the variable

void setGlobalSignedNumber(QString name, quint32 number)

Sets a unsigned number in the global signed number map.
Arguments:

- name: Name of the variable
- number: The number

QList<quint32> getGlobalSignedNumber(QString name, bool removeValue=false)

Returns a unsigned number from the global signed number map.
Arguments:

- name: Name of the variable
- removeValue: True if the variable shall be removed from the signed number map

Return: The first element in the result list is the result status (1=name found, 0=name not found). The second element is the read value.
Example:

```
//Read the stored value
var resultArray = scriptThread.getGlobalSignedNumber("S_Number");
if(resultArray[0] == 1)
{
	scriptThread.appendTextToConsole("snumber: " + resultArray[1]);
}
```

void globalUnsignedChangedSignal(QString name, quint32 number)

This signal is emitted if a signed number in the global signed number map has been changed. Scripts can connect a function to this signal.
Arguments:

- name: Name of the variable
- number: The value of the variable

void setGlobalRealNumber(QString name, double number)

Sets a real number in the global real number map.
Arguments:

- name: Name of the variable
- number: The number

QList<double> getGlobalRealNumber(QString name, bool removeValue=false)

Returns a real number from the global real number map.
Arguments:

- name: Name of the variable
- removeValue: True if the variable shall be removed from the real number map

Return: The first element in the result list is the result status (1.0=name found, 0.0=name not found). The second element is the read value.
Example:

```
//Read the stored value
var resultArray = scriptThread.getGlobalRealNumber("R_Number");
if(resultArray[0] == 1.0)
{
```

```
        scriptThread.appendTextToConsole("rnumber: " + resultArray[1]);
}
```

void globalRealChangedSignal(QString name, double number)

This signal is emitted if a real number in the global real number map has been changed. Scripts can connect a function to this signal.

Arguments:

- name: Name of the variable
- number: The value of the variable

Process

This chapter contains process related functions.

bool createProcessDetached(QString program, QStringList arguments, QString workingDirectory)

Starts a program with the given arguments in a new process, and detaches from it. Returns true on success, otherwise returns false. If the calling process exits, the detached process will continue to run unaffected. The process will be started in the directory workingDirectory. If workingDirectory is empty, the working directory is inherited from the calling process.

Arguments:

- program: The program to start
- arguments: The arguments
- workingDirectory: The working directory.

Return: True on success

Example:

```
var arguments = Array("D:/stylers.xml","D:/session.xml");
scriptThread.createProcessDetached("D:/notepad.exe", arguments,"");
```

int createProcess (QString program, QStringList arguments)

Starts a program with the given arguments in a new process, waits for it to finish, and then returns the exit code of the process. The environment and working directory are inherited from the calling process.

Arguments:

- program: The program to start
- arguments: The arguments

Return: True on success

Example:

```
var arguments = Array("D:/stylers.xml","D:/session.xml");
var result = scriptThread.createProcess("D:/notepad.exe", arguments);
```

ScriptProcess createProcessAsynchronous(QString program, QStringList arguments, int startWaitTime=30000, QString workingDirectory="")

Starts the program program with the arguments arguments in a new process. Any data the new process writes to the console is forwarded to the return process object. The environment and working directory are inherited from the calling process.

Note: Blocks until the process has been created or until startWaitTime milliseconds have passed (-1=infinite).

Arguments:

- program: The program to start
- arguments: The arguments
- startWaitTime: The max. wait time.
- workingDirectory: The working directory. If empty then the working directory of the process is set the ScriptCommunicator directory.

Return: The created process on success. An invalid object on failure.
Example:

```
var arguments = Array("D:/stylers.xml","D:/session.xml");
var process = scriptThread.createProcessAsynchronous("D:/notepad.exe", arguments);
if(typeof process == 'undefined')
{
	scriptThread.appendTextToConsole("could not start: D:/notepad.exe");
}
else
{
	scriptThread.waitForFinishedProcess(process);
}
```

bool waitForFinishedProcess(ScriptProcess process, int waitTime=30000)

Blocks until the process has finished or until msecs milliseconds have passed (-1=infinite).
Arguments:

- process: The process
- waitTime: The max. wait time.

Return: True if the process is finished.

int getProcessExitCode(ScriptProcess process)

Returns the exit code of process.
Arguments:

- process: The process

Return: The process exit code.

void killProcess(ScriptProcess process)

Kills the current process, causing it to exit immediately.
Arguments:

- process: The process

void terminateProcess(ScriptProcess process)

Attempts to terminate the process. The process may not exit as a result of calling this function (it is given the chance to prompt the user for any unsaved files, etc).
Arguments:

- process: The process

bool writeToProcessStdin(ScriptProcess process, QVector<unsigned char> data, int waitTime=30000)

Write data to the standard input of process. Returns true on success.
Note: Blocks until the writing is finished or until msecs milliseconds have passed (-1=infinite).
Arguments:

- program: The program to start
- data: The data
- waitTime: The max. wait time.

Return: True on success.

Example:

```
var arguments = Array("D:/stylers.xml","D:/session.xml");
var process = scriptThread.createProcessAsynchronous("D:/notepad.exe", arguments);
if(typeof process == 'undefined')
{
	scriptThread.appendTextToConsole("could not start: D:/notepad.exe");
}
else
{
	var data = scriptThread.stringToArray("test standard in\n");
	scriptThread.writeToProcessStdin(process, data);
}
```

QVector<unsigned char> readAllStandardOutputFromProcess(ScriptProcess process, bool isBlocking=false, quint8 blockByte='\n', qint32 blockTime=30000)

This function returns all data available from the standard output of process.

Note: If isBlocking is true then this function blocks until the blockByte has been received or blockTime has elapsed (-1=infinite).

Arguments:

- program: The program to start
- isBlocking: True if this function is blocking.
- BlockByte: The block byte.
- blockTime: The max. block time.

Return: The read data.

Example:

```
var arguments = Array("D:/stylers.xml","D:/session.xml");
var process = scriptThread.createProcessAsynchronous("D:/notepad.exe", arguments);
if(typeof process == 'undefined')
{
	scriptThread.appendTextToConsole("could not start: D:/notepad.exe");
}
else
{
	var res = scriptThread.readAllStandardOutputFromProcess(process, true,
				String('\n').charCodeAt(0), 10000);
	res = scriptThread.byteArrayToString(res).replace("\n", "")
	scriptThread.appendTextToConsole("stdout data:  " +  res);
}
```

QVector<unsigned char> readAllStandardErrorFromProcess(ScriptProcess process, bool isBlocking=false, quint8 blockByte='\n', qint32 blockTime=30000)

This function returns all data available from the standard error of process.

Note: If isBlocking is true then this function blocks until the blockByte has been received or blockTime has elapsed (-1=infinite).

Arguments:

- program: The program to start
- isBlocking: True if this function is blocking.
- BlockByte: The block byte.

- blockTime: The max. block time.

Return: The read data.

Example:

```
var arguments = Array("D:/stylers.xml","D:/session.xml");
var process = scriptThread.createProcessAsynchronous("D:/notepad.exe", arguments);
if(typeof process == 'undefined')
{
	scriptThread.appendTextToConsole("could not start: D:/notepad.exe");
}
else
{
	var res = scriptThread.readAllStandardErrorFromProcess(process, true,
				String('\n').charCodeAt(0), 10000);
	res = scriptThread.byteArrayToString(res).replace("\n", "")
	scriptThread.appendTextToConsole("stdout data:  " +  res);
}
```

Miscellaneous

This chapter contains general functions.

QStringList availableSerialPorts(void)

Returns a list with the name of all available serial ports.

Example:

```
var availablePorts = scriptThread.availableSerialPorts();
for(var i = 0; i < availablePorts.length; i++)
{
	UI_serialPortInfoListBox.addItem(availablePorts[i]);
}
```

bool setScriptState(quint8 state, QString scriptTableEntryName)

Sets the state of a script (running, paused or stopped).

Note: The script must be in the script table (script window) and a script can not set it's own state.

Arguments:

- state: The state in which the script shall be switched. Possible value are:
  - 0: running
  - 1: paused
  - 2: stopped
- scriptTableEntryName: The name of the script in the script-table (script window).

Return: True if scriptTableEntryName has been found in the script-table and the state has a valid value.

Example:

```
scriptThread.setScriptState(0, "send input");//Start the script.
scriptThread.sleepFromScript(3000);

scriptThread.setScriptState(1, "send input");//Pause the script.
scriptThread.sleepFromScript(3000);

scriptThread.setScriptState(2, "send input");//Stop the script.
```

QString getScriptTableName(void)

Returns the script-table (script window) name of the calling script.

void appendTextToConsole(QString text, bool newLine=true)

This function can be used to append a text to the script window console.

Arguments:

- text: The text which has to be appended to the console
- newLine: If true then the text will be appended in a new line

Example:

```
scriptThread.appendTextToConsole("exception in dataReceivedSlot: " + e);
```

void sleepFromScript (quint32 timeMs)

Forces the script thread to sleep for ms milliseconds.

Arguments:

- timeMs: The time to sleep in milliseconds

Example:

```
scriptThread.sleepFromScript(10);
```

bool scriptShallExit(void)

Returns true if the script shall exit. This can occur if:

- the user has pressed the stop button (main or script window)
- the script has called scriptThread.stopScript()
- an uncaught exception has been occurred

QString byteArrayToString (QVector<unsigned char> data)

Converts a byte array which contains ASCII characters into an ASCII string.

Example:

```
var array = Array(48, 49, 50);
var string = scriptThread.byteArrayToString(array);
```

QString byteArrayToHexString (QVector<unsigned char> data)

Converts a byte array into a hex string.

Example:

```
var array = Array(2, 3, 4, 33);
var string = scriptThread.byteArrayToHexString(array);
```

QVector<unsigned char> stringToArray(QString str)

Converts an ASCII string into a byte array.

Example:

```
var array = scriptThread.stringToArray("Test");
```

QVector<unsigned char> addStringToArray(QVector<unsigned char> array, QString str)

Adds an ASCII string to a byte array.

Example:

```
var array = Array(0,1,2,3,4);
array = scriptThread.addStringToArray(array, "Test");
```

QTimer createTimer (void)

Creates a timer object (for more details about the script timer see chapter Script timer class).

Return: The created timer object

Example:

```
var timer = scriptThread.createTimer();
```

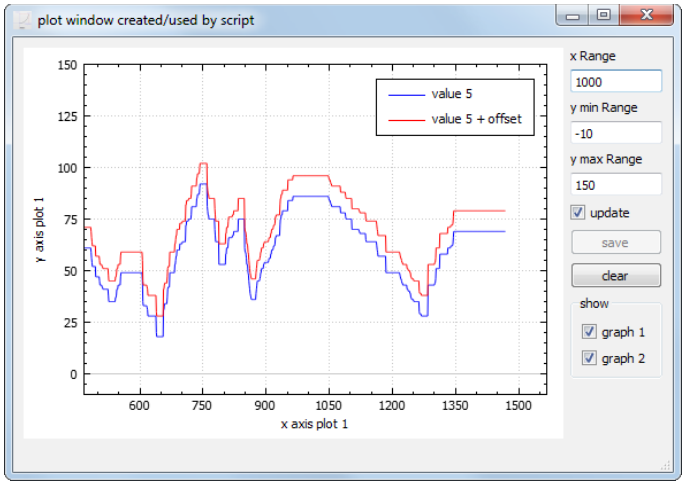
ScriptPlotWindow createPlotWindow (void)

Creates a plot window object (for more details about the script plot window see chapter Script plot window class).

Return: The created plot window object

Example:

```
var plotWindow = scriptThread.createPlotWindow();
```



bool loadScript(QString scriptPath, bool isRelativePath=true)

Loads/includes one script (QtScript has no built-in include mechanism).

Arguments:

- scriptPath: The file path
- isRelativePath: True if the path of script is relative to current script (which executes this function)

Return: True on success

Example:

```
var result = scriptThread.loadScript("Testscript.js");
```

bool loadLibrary(QString path, bool isRelativePath=true)

Loads a dynamic link library and calls the init function (void init(QScriptEngine* engine)).

With this function scripts can extend their functionality. For further information see chapter Dynamic link libraries.

Arguments:

- path: The library file path
- isRelativePath: True if the library path is relative to the current script (which executes this function)

Return: True on success

Example:

```
var result = scriptThread.loadLibrary("testDll.dll");
```

bool loadUserInterfaceFile(QString path, bool isRelativePath=true, bool showAfterLoading = true)

Loads a user interface file and shows the GUI.

Arguments:

- path: The user interface file path.
- isRelativePath: True if the file path is relative to the current script (which executes this function)
- showAfterLoading: True if the first element of the user interface file (normally a window) shall be shown

Return: True on success

Example:

```
if(!uiFileLoaded)
{
	scriptThread.appendTextToConsole("no ui file in script window, loading file:
					guiExample.ui");
	uiFileLoaded = scriptThread.loadUserInterfaceFile("guiExample.ui");
}
```

void stopScript(void)

This function stops the current script.

Example:

```
if(error)
{
	scriptThread.stopScript(scriptThread);
}
```

QStringList getLocalIpAdress(void)

Returns all IP addresses (IPv4 and IPv6) found on the host machine (array with strings).

Example:

```
var ipList = scriptThread.getLocalIpAdress();
for(var i = 0; i < ipList.length; i++)
{
	 scriptThread.appendTextToConsole(ipList[i])
}
```

bool showReceivedDataInConsoles(bool show)

Scripts can switch on/off the adding of received data in the main window consoles (for fast data transfers).

Arguments:

- show: True=show received data

Return: The old value (on/off)

bool showTransmitDataInConsoles(bool show)

Scripts can switch on/off the adding of transmitted data in the main window consoles (for fast data transfers).

Arguments:

- show: True=show transmitted data

Return: The old value (on/off)

void addMessageToLogAndConsoles(QString text, bool forceTimeStamp=false)

Adds a message into the logs and the main window consoles (if they are active).

Arguments:

- text: The message text

- forceTimeStamp: True if a time stamp shall be generated (independently from the time stamp settings)

bool setScriptThreadPriority(QString priority)

Sets the priority of the script thread (which executes the current script).
Note: Per default script threads have 'LowestPriority'.
Arguments:

- priority: The new priority. Possible values are:
  - LowestPriority
  - LowPriority
  - NormalPriority
  - HighPriority
  - HighestPriority

Return: True on success.
Example:

```
scriptThread.setScriptThreadPriority("NormalPriority");
```

QString getCurrentVersion

Returns the current version of ScriptCommunicator (string) .
Version format: major.minor (eg. 3.09)

QString exitScriptCommunicator

This function exits ScriptCommunicator.

void setBlockTime(quint32 blockTime)

Sets the script block time (ms).
Note: If the user presses the stop button the script must be exited after this time. If not then the script is regarded as blocked and will be terminated. The default is 5000.

QString currentCpuArchitecture(void)

Returns the architecture of the CPU that the application is running on, in text format.
Note that this function depends on what the OS will report and may not detect the actual CPU architecture if the OS hides that information or is unable to provide it. For example, a 32-bit OS running on a 64-bit CPU is usually unable to determine the CPU is actually capable of running 64-bit programs.
Values returned by this function are mostly stable: an attempt will be made to ensure that they stay constant over time and match the values returned by QSysInfo::builldCpuArchitecture(). However, due to the nature of the operating system functions being used, there may be discrepancies.

Typical returned values are (note: list not exhaustive):

- "arm"
- "arm64"
- "i386"
- "ia64"
- "mips"
- "mips64"

- "power"
- "power64"
- "sparc"
- "sparcv9"
- "x86_64

QString productType(void)

Returns the product name of the operating system this application is running in. If the application is running on some sort of emulation or virtualization layer (such as WINE on a Unix system), this function will inspect the emulation / virtualization layer.
Values returned by this function are stable and will not change over time, so applications can rely on the returned value as an identifier, except that new OS types may be added over time.

Linux and Android note: this function returns "android" for Linux systems running Android userspace, notably when using the Bionic library. For all other Linux systems, regardless of C library being used, it tries to determine the distribution name and returns that. If determining the distribution name failed, it returns "unknown".

BlackBerry note: this function returns "blackberry" for QNX systems running the BlackBerry userspace, but "qnx" for all other QNX-based systems.

Darwin, OS X and iOS note: this function returns "osx" for OS X systems, "ios" for iOS systems and "darwin" in case the system could not be determined.

FreeBSD note: this function returns "debian" for Debian/kFreeBSD and "unknown" otherwise.

Windows note: this function returns "winphone" for builds for Windows Phone, "winrt" for WinRT builds, "wince" for Windows CE and Embedded Compact builds, and "windows" for normal desktop builds.

For other Unix-type systems, this function usually returns "unknown".

QString productVersion(void)

Returns the product version of the operating system in string form. If the version could not be determined, this function returns "unknown".
It will return the Android, BlackBerry, iOS, OS X, Windows full-product versions on those systems. In particular, on OS X, iOS and Windows, the returned string is similar to the macVersion() or windowsVersion() enums.

On Linux systems, it will try to determine the distribution version and will return that. This is also done on Debian/kFreeBSD, so this function will return Debian version in that case.

In all other Unix-type systems, this function always returns "unknown".

Note: The version string returned from this function is only guaranteed to be orderable on Android, BlackBerry, OS X and iOS. On Windows, some Windows versions are text ("XP" and "Vista", for example). On Linux, the version of the distribution may jump unexpectedly, please refer to the distribution's documentation for versioning practices.

QStringList getScriptArguments(void)

Returns the script arguments (command-line argument -A).
Example:

```
var args = scriptThread.getScriptArguments();
for(var i = 0; i < args.length; i++)
```

```
{
        scriptThread.appendTextToConsole(args[i])
}
```

QString getScriptCommunicatorFolder(void)

Returns the ScriptCommunicator program folder.

QString getUserDocumentsFolder(void)

Returns the directory containing user document files.

bool addTabsToMainWindow(ScriptTabWidget tabWidget)*

Adds script tabs to the main window (all tabs are removed from tabWidget).
Note: This function fails in command-line mode.
Arguments:

- tabWidget: The ScriptTabWidget which contains the tabs which shall be added to the main window

Return: True on success
Example:

```
scriptThread.loadUserInterfaceFile("tabs.ui", true, false);

//Remove the tab from the dialog and add it to the main window.
if(!scriptThread.addTabsToMainWindow(UI_TabWidget))
{
       scriptThread.messageBox("Critical", "Error", "addTabsToMainWindow failed");
       scriptThread.stopScript();
}
```

An example can be found under exampleScripts\WorkerScripts\TestAddTabsToMainWindow.

bool addToolBoxPagesToMainWindow(ScriptToolBox scriptToolBox)*

Adds script toolbox pages to the main window (all pages are removed from scriptToolBox).
Note: This function fails in command-line mode.
Arguments:

- scriptToolBox: The ScriptToolBox which contains the pages which shall be added to the main window

Return: True on success
Example:

```
scriptThread.loadUserInterfaceFile("pages.ui", true, false);

//Remove the pages from the dialog and add it to the main window.
if(!scriptThread.addToolBoxPagesToMainWindow(UI_ToolBox))
{
       scriptThread.messageBox("Critical", "Error", "addToolBoxPagesToMainWindow
                               failed");
       scriptThread.stopScript();
}
```

An example can be found under
exampleScripts\WorkerScripts\TestAddToolBoxPageToMainWindow.

Script UDP socket class

This class is a wrapper class for the Qt class QUdpSocket. With this class the script can send and receive data (additional to the main interface) with a UDP socket. To create a UDP socket object the function ScriptUdpSocket createUdpSocket(void) must be used.

The functions and signals which can be used are described in the following chapters.

bool bind(quint16 port)

Binds the socket to a port.
Arguments:

- port: The port

bool isOpen(void)

Returns true if the UDP socket is open/listening.

void close(void)

Closes the socket.

bool hasPendingDatagrams(void)

Returns true if a received datagram can be read from the socket.

QVector<unsigned char> readDatagram(void)

Returns the data from one received datagram.

QVector<unsigned char> readAll(void)

Reads all received datagrams (the data from the single datagrams are inserted in one unsigned char vector)

quint64 write(QVector<unsigned char> data, QString hostAdress, quint16 hostPort)

Writes data to the socket.
Arguments:

- data: The data
- hostAdress: The host address
- hostPort: The port of the host

Return: The number of written bytes

quint64 writeString(QString string, QString hostAdress, quint16 hostPort)

Write a string to the socket.
Arguments:

- string: The string
- hostAdress: The host address
- hostPort: The port of the host

Return: The number of written bytes

void enableMainInterfaceRouting(QString routingHostAddress, quint16 routingHostPort)

Enables the main interface routing (all data from the main interface is send with this socket and

all received (with this socket) data is sent with the main interface).
Arguments:

- routingHostAdress: The host address, to which all data from the main interface is sent
- routingHostPort: The port of the host, to which all data from the main interface is sent

void disableMainInterfaceRouting(void)

Disables the main interface routing.

bool canReadLine(void)

This function checks if a data line (ends with EOL ('\n')) is ready to be read.
Return: True if a line is ready to be read.

QString readLine(bool removeNewLine=true, bool removeCarriageReturn=true)

This function reads a line (a line ends with a '\n') of ASCII characters.
Arguments:

- removeNewLine: If removeNewLine is true then the '\n' will not returned (is removed from the received line).
- removeCarriageReturn: If removeCarriageReturn is true then a '\r' in front of '\n' will also not returned.

Return: The received line. If no new data line is ready for reading this functions returns an empty string.
Example:

```
function readyReadSlot()
{
	if(udpSocket.canReadLine())
	{
		var line = udpSocket.readLine();
		scriptThread.appendTextToConsole("line received:  "  + line);
	}
}
```

QStringList readAllLines(bool removeNewLine=true, bool removeCarriageReturn=true)

This function reads all available lines (a line ends with a '\n') of ASCII characters.
Arguments:

- removeNewLine: If removeNewLine is true then the '\n' will not returned (is removed from the received lines).
- removeCarriageReturn: If removeCarriageReturn is true then a '\r' in front of '\n' will also not returned.

Return: The received lines. If no new data line is ready for reading this functions returns an empty list.
Example:

```
function readyReadSlot()
{
	var lines = udpSocket.readAllLines();
	for(var index = 0; index < lines.length; index++)
	{
		scriptThread.appendTextToConsole("line received:  "  + lines[index]);
	}
}
```

void readyReadSignal(void)

This signal is emitted if data can be read from the socket (if a datagram has been received). Scripts can connect a function to this signal.

UDP socket example

The following Code shows the typically use of the UDP socket class:

```
function readyReadSlot()
{
	var data = udpSocket.readAll();
	udpSocket.write(data, "127.0.0.1", 111);
}
var udpSocket = scriptThread.createUdpSocket();

//connect the readyReadSlot function to the readyReadSignal signal
udpSocket.readyReadSignal.connect(readyReadSlot);
udpSocket.bind(11112);
```

Script TCP client class

This class is a wrapper class for the Qt class QTcpSocket. With this class the script can send and receive data (additional to the main interface) with a TCP socket. To create a TCP socket object the function ScriptTcpClient createTcpClient (void) must be used.

The functions and signals which can be used are described in the following chapters.

void connectToHost(QString hostAdress, quint16 port)

This function connects the socket to a TCP server.
Arguments:

- hostAdress: The host address
- port: The port of the host

bool isOpen(void)

Returns true if the TCP client is open/connected.

void close(void)

This function closes the socket.

bool isReadable(void)

Returns true if data can be read from the socket.

quint64 bytesAvailable(void)

Returns the number of bytes which are available for reading.

QVector<unsigned char> readAll(void)

This function returns all received bytes.

quint64 write(QVector<unsigned char> data)

Writes data to the socket.
Arguments:

- data: The data

Return: The number of written bytes

quint64 writeString(QString string)

Writes a string to the socket.

Arguments:

- string: The string

Return: The number of written bytes

QString getErrorString(void)

Returns a human-readable description of the last error that has been occurred.

void enableMainInterfaceRouting(void)

Enables the main interface routing (all data from the main interface is send with this socket and all received (with this socket) data is sent with the main interface).

void disableMainInterfaceRouting(void)

Disables the main interface routing.

void setProxy(QString proxyType = "NO_PROXY", QString proxyUserName= "", QString proxyPassword = "", QString proxyIpAddress = "", quint16 proxyPort = 0)

Sets the network proxy.

Arguments:

- proxyType: The proxy type, possible values are:
  - NO_PROXY
  - SYSTEM_PROXY
  - CUSTOM_PROXY
- proxyUserName: The proxy user name
- proxyPassword: The proxy user password
- proxyIpAddress: The proxy ip address
- proxyPort: The proxy port

Example:

```
var tcpClient = scriptThread.createTcpClient();
//No proxy.
tcpClient.setProxy("NO_PROXY");

//Use the system proxy settings.
tcpClient.setProxy("SYSTEM_PROXY", "proxy_user", "proxy_password");

//Use custom proxy settings.
tcpClient.setProxy("CUSTOM_PROXY", "proxy_user", "proxy_password""169.254.224.120",
83);
```

bool canReadLine(void)

This function checks if a data line (ends with EOL ('\n')) is ready to be read.

Return: True if a line is ready to be read.

QString readLine(bool removeNewLine=true, bool removeCarriageReturn=true)

This function reads a line (a line ends with a '\n') of ASCII characters.
Arguments:

- removeNewLine: If removeNewLine is true then the '\n' will not returned (is removed from the received line).
- removeCarriageReturn: If removeCarriageReturn is true then a '\r' in front of '\n' will also not returned.

Return: The received line. If no new data line is ready for reading this functions returns an empty string.
Example:

```
function readyReadSlot()
{
	if(tcpClient.canReadLine())
	{
		var line = tcpClient.readLine();
		scriptThread.appendTextToConsole("line received:  "  + line);
	}
}
```

QStringList readAllLines(bool removeNewLine=true, bool removeCarriageReturn=true)

This function reads all available lines (a line ends with a '\n') of ASCII characters.
Arguments:

- removeNewLine: If removeNewLine is true then the '\n' will not returned (is removed from the received lines).
- removeCarriageReturn: If removeCarriageReturn is true then a '\r' in front of '\n' will also not returned.

Return: The received lines. If no new data line is ready for reading this functions returns an empty list.
Example:

```
function readyReadSlot()
{
	var lines = tcpClient.readAllLines();
	for(var index = 0; index < lines.length; index++)
	{
		scriptThread.appendTextToConsole("line received:  "  + lines[index]);
	}
}
```

void connectedSignal (void)

This signal is emitted if the connection has been established. Scripts can connect a function to this signal.

void disconnectedSignal (void)

This signal is emitted if the connection has been disconnected. Scripts can connect a function to this signal.

void readyReadSignal(void)

This signal is emitted if data can be read from the socket. Scripts can connect a function to this signal.

void errorSignal(int error)

This signal is emitted after an error has been occurred. The error parameter describes the type of error that has been occurred.

Arguments:

- error: The error

Internally error has the type QabstractSocket::SocketError. Therefore error can have the following values:

QAbstractSocket::ConnectionRefusedError	0	The connection was refused by the peer (or timed out).
QAbstractSocket::RemoteHostClosedError	1	The remote host closed the connection. Note that the client socket (i.e., this socket) will be closed after the remote close notification has been sent.
QAbstractSocket::HostNotFoundError	2	The host address was not found.
QAbstractSocket::SocketAccessError	3	The socket operation failed because the application lacked the required privileges.
QAbstractSocket::SocketResourceError	4	The local system ran out of resources (e.g., too many sockets).
QAbstractSocket::SocketTimeoutError	5	The socket operation timed out.
QAbstractSocket::DatagramTooLargeError	6	The datagram was larger than the operating system's limit (which can be as low as 8192 bytes).
QAbstractSocket::NetworkError	7	An error occurred with the network (e.g., the network cable was accidentally plugged out).
QAbstractSocket::AddressInUseError	8	The address specified to QAbstractSocket::bind() is already in use and was set to be exclusive.
QAbstractSocket::SocketAddressNotAvailableError	9	The address specified to QAbstractSocket::bind() does not belong to the host.
QAbstractSocket::UnsupportedSocketOperationError	10	The requested socket operation is not supported by the local operating system (e.g., lack of IPv6 support).
QAbstractSocket::ProxyAuthenticationRequiredError	12	The socket is using a proxy, and the proxy requires authentication.
QAbstractSocket::SslHandshakeFailedError	13	The SSL/TLS handshake failed, so the connection was closed (only used in QSslSocket)
QAbstractSocket::UnfinishedSocketOperationError	11	Used by QAbstractSocketEngine only, The last operation attempted has not finished yet (still in progress in the background).
QAbstractSocket::ProxyConnectionRefusedError	14	Could not contact the proxy server because the connection to that server was denied
QAbstractSocket::ProxyConnectionClosedError	15	The connection to the proxy server was closed unexpectedly (before the connection to the final peer was established)

QAbstractSocket::ProxyConnectionTimeoutError	16	The connection to the proxy server timed out or the proxy server stopped responding in the authentication phase.
QAbstractSocket::ProxyNotFoundError	17	The proxy address set with setProxy() (or the application proxy) was not found.
QAbstractSocket::ProxyProtocolError	18	The connection negotiation with the proxy server failed, because the response from the proxy server could not be understood.
QAbstractSocket::OperationError	19	An operation was attempted while the socket was in a state that did not permit it.
QAbstractSocket::SslInternalError	20	The SSL library being used reported an internal error. This is probably the result of a bad installation or misconfiguration of the library.
QAbstractSocket::SslInvalidUserDataError	21	Invalid data (certificate, key, cypher, etc.) was provided and its use resulted in an error in the SSL library.
QAbstractSocket::TemporaryError	22	A temporary error occurred (e.g., operation would block and socket is non-blocking).
QAbstractSocket::UnknownSocketError	-1	An unidentified error occurred.

TCP client example

The following Code shows the typically use of the TCP client class:

```
//Connection established.
function connectSlot()
{
	var array = Array(1,2,3,4,5,6);
	//send data to the host
	tcpClient.write(array);
}
//Connection closed.
function disconnectSlot()
{
	scriptThread.appendTextToConsole("disconnected ");
	tcpClient.close();
}
//Data has been received.
function readyReadSlot ()
{
	var array = tcpClient.readAll();
	scriptThread.appendTextToConsole("data received:  "  + array);
	//send data to the host
	tcpClient.write(array);

}
//An error has been ocurred.
function tcpClientErrorSlot(error)
{
	if(error != 1)
	{//The error is not QAbstractSocket::RemoteHostClosedError.
		disconnectSlot();
	}
```

```
        if(error == 0) //QAbstractSocket::ConnectionRefusedError
        {
                scriptThread.messageBox("Critical", "TCP error",
                                        "The connection was refused. " +
                                        "Make sure the server is running, " +
                                        "and check that the host name and port " +
                                        "settings are correct.");
        }
        else if(error == 1) //QAbstractSocket::RemoteHostClosedError
        {
                //The connection has been closed. Do nothing.
        }
        else if(error == 2) //QAbstractSocket::HostNotFoundError
        {
                scriptThread.messageBox("Critical", "TCP error",
                                        "The server was not found. Please check the " +
                                        "host name and port settings.");
        }
        else
        {
                scriptThread.messageBox("Critical", "TCP error",
                                        "The following error occurred: " +
                                        tcpClient.getErrorString());
        }
}

var tcpClient = scriptThread.createTcpClient();
tcpClient.readyReadSignal.connect(readyReadSlot);
tcpClient.disconnectedSignal.connect(disconnectSlot);
tcpClient.connectedSignal.connect(connectSlot);
tcpClient.errorSignal.connect(tcpClientErrorSlot);
tcpClient.connectToHost("127.0.0.1", 111);
```

Script TCP server class

This class is a wrapper class for the Qt class QTcpServer. With this class the script can create a TCP server. To create a TCP server object the function ScriptTcpServer createTcpServer (void) must be used.

The functions and signals which can be used are described in the following chapters.

bool listen(quint16 port)

Call this function to start listening for new connections.
Arguments:

- port: The port for listening

Return: True on success

bool isListening (void)

Returns true if the socket is listening for new connections.

void setMaxPendingConnections(int numConnections)

Set the max. pending connections.
Arguments:

- numConnections: The max. pending connections

int maxPendingConnections(void)
Returns the max. pending connections.

void close(void)
This function closes the TCP server.

ScriptTcpClient nextPendingConnection(void)
Return the next pending connection (returns a script TCP client).

void newConnectionSignal (void)
This signal is emitted if a new connection has been established. Scripts can connect a function to this signal.

TCP server example
The following Code shows the typically use of the TCP server class:

```
function connectionEstablished()
{
	tcpServerClient = tcpServer.nextPendingConnection();
	tcpServerClient.disconnectedSignal.connect(disconnectSlot);
	tcpServerClient.readyReadSignal.connect(readyReadSlot);
}
function disconnectSlot()
{
	scriptThread.appendTextToConsole("disconnected ");
}
function readyReadSlot()
{
	var array = tcpServerClient.readAll();
	scriptThread.appendTextToConsole("data received:  "  + array);
	//send data to the host
	tcpServerClient.write(array);
}
var tcpServer = scriptThread.createTcpServer();
var tcpServerClient = undefined;
tcpServer.newConnectionSignal.connect(connectionEstablished);
tcpServer.listen(112);
```

Script serial port class
This class is a wrapper class for the Qt class QSerialPort. With this class the script can send and receive data (additional to the main interface) with a serial port. To create a serial port socket object the function ScriptSerialPort createSerialPort (void) must be used.

The functions and signals which can be used are described in the following chapters.

Note:

If the script serial port is connected the RTS (request to send) and DTR (data terminal ready) signals are automatically set to high (and set low at disconnect).

void setPortName(const QString name)
Sets the serial port name.
Arguments:

- name: The name of the serial port

QString portName(void)
Returns the serial port name.

bool setBaudRate(qint32 baudRate)
Sets the baudrate.
Arguments:
- baudRate: The new baud rate

Return: True on success

qint32 baudRate(void)
Returns the baudrate.

bool setDataBits(quint32 dataBits)
Sets the number of data bits.
Arguments:
- dataBits: The new number of data bits

Return: True on success

quint32 dataBits(void)
Returns the number of data bits.

bool setParity(QString parityString)
Sets the parity.
Arguments:
- parityString: The new parity. Possible values are: "None ", "Even ", "Odd ", "Space" and "Mark"

Return: True on success

QString parity(void)
Returns the parity. Possible values are: "None ", "Even", "Odd ", "Space" and "Mark".

bool setStopBits(QString stopBitsString)
Sets the number of stop bits.
Arguments:
- stopBitsString: The new number of stop bits. Possible values are: "1 ", "1.5" and "2"

Return: True on success

QString stopBits (void)
Returns the number of stop bits. Possible values are: "1 ", "1.5" and "2".

bool setFlowControl(QString flowString)
Sets the flow control.
Arguments:
- flowString: The new flow control. Possible values are: "RTS/CTS", "XON/XOFF" and "None"

Return: True on success

QString flowControl (void)
Returns the flow control. Possible values are: "RTS/CTS ", " XON/XOFF" and "None".

QString errorString(void)
Returns the error string from the serial port (contains additional information in the case of an error).

bool open(void)
Opens the serial port.
Return: True on success

void close(void)
Closes the serial port.

void setDTR(bool set)
Sets the DTR (data terminal ready) pin.
Note: The default for the DTR (data terminal ready) pin is 1 (if the serial port is completely configured and open).
Arguments:

- set: true=set the pin to 1, false=set the pin to 0

void setRTS(bool set)
Sets the RTS (request to send) pin.
Note: The default for the RTS (request to send) pin is 0. This call has only an effect if the serial port is completely configured and open.
Arguments:

- set: true=set the pin to 1, false=set the pin to 0

bool isOpen(void)
Returns true if the serial port is open.

quint64 bytesAvailable(void)
Returns the number of bytes which are available for reading.

QVector<unsigned char> readAll(void)
This function returns all received bytes.

quint64 write(QVector<unsigned char> data)
Writes data to the serial port.
Arguments:

- data: The data

Return: The number of written bytes

quint64 writeString(QString string)
Writes a string to the serial port.

Arguments:

- string: The string

Return: The number of written bytes

qint64 bytesToWrite(void)

Returns the number of bytes which are not written yet.

bool waitForBytesWritten(int msecs)

This function waits until all bytes have been written (sent) or the time in msec has been elapsed.

Arguments:

- msecs: The max. time to wait

Return: True if all bytes have been written

void enableMainInterfaceRouting(void)

Enables the main interface routing (all data from the main interface is send with this socket and all received (with this socket) data is sent with the main interface).

void disableMainInterfaceRouting(void)

Disables the main interface routing.

bool canReadLine(void)

This function checks if a data line (ends with EOL ('\n')) is ready to be read.

Return: True if a line is ready to be read.

QString readLine(bool removeNewLine=true, bool removeCarriageReturn=true)

This function reads a line (a line ends with a '\n') of ASCII characters.

Arguments:

- removeNewLine: If removeNewLine is true then the '\n' will not returned (is removed from the received line).
- removeCarriageReturn: If removeCarriageReturn is true then a '\r' in front of '\n' will also not returned.

Return: The received line. If no new data line is ready for reading this functions returns an empty string.

Example:

```
function readyReadSlot()
{
	if(serialPort.canReadLine())
	{
		var line = serialPort.readLine();
		scriptThread.appendTextToConsole("line received:  "  + line);
	}
}
```

QStringList readAllLines(bool removeNewLine=true, bool removeCarriageReturn=true)

This function reads all available lines (a line ends with a '\n') of ASCII characters.

Arguments:

- removeNewLine: If removeNewLine is true then the '\n' will not returned (is removed from the received lines).
- removeCarriageReturn: If removeCarriageReturn is true then a '\r' in front of '\n' will also not returned.

Return: The received lines. If no new data line is ready for reading this functions returns an empty list.

Example:

```
function readyReadSlot()
{
	var lines = serialPort.readAllLines();
	for(var index = 0; index < lines.length; index++)
	{
		scriptThread.appendTextToConsole("line received:  "  + lines[index]);
	}
}
```

void readyReadSignal(void)

This signal is emitted if data is available for reading (if data has been received). Scripts can connect a function to this signal.

quint32 getSerialPortSignals(void)

Returns the state of the serial port signals (pins).

The signals are bit coded:

- NoSignal = 0x00,
- DataTerminalReadySignal = 0x04
- DataCarrierDetectSignal = 0x08
- DataSetReadySignal = 0x10
- RingIndicatorSignal = 0x20
- RequestToSendSignal = 0x40
- ClearToSendSignal = 0x80

Script serial port example

The following Code shows the typically use of the script serial port class:

```
function readyReadSlot()
{
	var array = serialPort.readAll();
	scriptThread.appendTextToConsole("data received:  "  + array);
	//send data to the host
	serialPort.write(array);
}

var serialPort = scriptThread.createSerialPort();
serialPort.readyReadSignal.connect(readyReadSlot);

serialPort.setPortName("COM1");
serialPort.setDTR(false);
if (serialPort.open())
{
	serialPort.setBaudRate(19200);
	serialPort.setDataBits(8);
	serialPort.setParity("None");
	serialPort.setStopBits("1.5");
```

```
        serialPort.setFlowControl("None");
        serialPort.setDTR(true);
        serialPort.setRTS(true);
        scriptThread.appendTextToConsole("serial port signals:" +
                              serialPort.getSerialPortSignals().toString(16) );
}
else
{
        scriptThread.messageBox("Critical", 'error', 'could not open serial port');
}
```

Script cheetah SPI class

With this class the script can create a cheetah SPI interface (only SPI Master). To create a cheetah SPI interface object the function ScriptCheetahSpi createCheetahSpiInterface(void) must be used.

The functions and signals which can be used are described in the following chapters.

QString detectDevices(void)

Returns a string which contains informations about all detected devices.

bool connect(quint32 port, qint16 mode, quint32 baudrate)

Connects to a cheetah SPI interface.
Arguments:

- port: The interface port
- mode: The SPI mode (0-3)
- baudrate: The baudrate (kHz)

Return: True on success.

void disconnect(void

Disconnects from the cheetah SPI interface.

bool sendReceiveData(QVector<unsigned char> sendData, quint8 chipsSelect)

Sends and receive data with the cheetah SPI interface (the received data must be read with readAll).
Arguments:

- sendData: The data which shall be sent
- chipSelect: The number of the chip select which shall be activated (0-2)

Return: True on success.

QVector<unsigned char> readAll)

Returns all received data from the last sendReceiveData call.

Script cheetah SPI example

The following Code shows the typically use of the script cheetah SPI class:

```
var interface = scriptThread.createCheetahSpiInterface();
if(interface.connect(0, 1, 12000))
{       var data = Array(0xD7,0xff,0xff,0xff,0xff);
        if(interface.sendReceiveData(data, 0))
        {
                var receivedData = interface.readAll();
                scriptThread.appendTextToConsole("data received:  "  + receivedData);
```

```
        }
        else
        {
                scriptThread.messageBox("Critical", "error", 'sending failed');
        }
        interface.disconnect();
}
else
{
        scriptThread.messageBox("Critical", "error", 'could not connect to
                                interface');
}
```

Script PCAN class

With this class the script can create a PCAN interface. To create a PCAN interface object the function ScriptPcan createPcanInterface(void) must be used.

The functions and signals which can be used are described in the following chapters.

bool open(quint8 channel, quint32 baudrate, bool busOffAutoReset, bool powerSupply)

Opens a PCAN interface.
Arguments:

- channel: The PCAN channel
- baudRate: The baudrate (kHz), possible values are: 1000, 800, 500, 250, 125,100,95,83,50,47,33,20,10,5
- busOffAutoReset: Automatic reset on BUS-OFF on/off
- powerSupply: 5 volt power supply on/off

Return: True on success.

void close(void)

Closes the PCAN interface.

bool setFilter(bool filterExtended, quint32 filterFrom, quint32 filterTo)

Configures the reception filter.
Arguments:

- filterExtended: True if the filer message type is extended (29-bit identifier) or false if the filter message type is standard (11-bit identifier)
- filterFrom: The lowest CAN ID to be received
- filterTo: The highest CAN ID to be received

Return: True on success.

bool sendCanMessage(quint8 type, quint32 canId, QVector<unsigned char> data)

Sends a can message. If more then 8 data bytes are given several can messages with the same can id will be sent.
Arguments:

- type: The can message type: 0=standard, 1=standard remote-transfer-request, 2=extended, 3= extended remote-transfer-request
- canId: The CAN id
- data: The CAN data.

Return: True on success.

bool isConnected(void)
Returns true if connected to a PCAN interface.

QString getStatusString(void)
Returns the current status as string.

quint32 getCurrentStatus(void)
Returns the current status.

QList<quint8> getCanParameter(quint8 parameter)
Reads a PCAN parameter.
Arguments:

- parameter: The PCAN parameter. Possible values are:
    - 0x01=PCAN_DEVICE_NUMBER
    - 0x02=PCAN_5VOLTS_POWER
    - 0x07=PCAN_BUSOFF_AUTORESET
    - 0x08=PCAN_LISTEN_ONLY
    - 0x0F=PCAN_RECEIVE_STATUS
    - 0x10=PCAN_CONTROLLER_NUMBER
    - 0x15=PCAN_CHANNEL_IDENTIFYING

Return: Byte 0= status(1=success, 0=failure), Byte 1= the parameter value

bool setCanParameter(quint8 parameter, quint8 data)
Sets a PCAN parameter.
Arguments:

- parameter: The PCAN parameter. Possible values are:
    - 0x01=PCAN_DEVICE_NUMBER
    - 0x02=PCAN_5VOLTS_POWER
    - 0x07=PCAN_BUSOFF_AUTORESET
    - 0x08=PCAN_LISTEN_ONLY
    - 0x0F=PCAN_RECEIVE_STATUS
    - 0x15=PCAN_CHANNEL_IDENTIFYING
- data: The new parameter value

Return: True on success.

void canMessagesReceivedSignal(QVector<quint8> types, QVector<quint32> messageIds, QVector<quint32> timestamps, QVector<QVector<unsigned char>> data)
This signal is emitted if a can message (or several) has been received with the main interface. Scripts can connect a function to this signal.
Arguments:

- types: The can types of the received can messages (0=standard, 1=standard remote-transfer-request, 2=extended, 3= extended remote-transfer-request)
- messageIds: The can ids of the received can messages

- timestamps: The time-stamps of the received can messages
- data: The data of the received can messages

Script PCAN example

The following Code shows the typically use of the script PCAN class:

```
function pcanMessagesReceived(type, id, timeStamp, data)
{
	for(var index = 0; index < type.length; index++)
	{
		pcan.sendCanMessage(type[index], id[index], data[index], 0, 0);
	}
}
var pcan = scriptThread.createPcanInterface();
pcan.canMessagesReceivedSignal.connect(pcanMessagesReceived);

if(!pcan.open(2, 1000, true, false))
{
	scriptThread.messageBox("Critical", "error", 'could not open pcan');
}
if(!pcan.setFilter(true, 0, 0x1fffffffff))
{
	scriptThread.messageBox("Critical", "error", 'setFilter failed');
}

//PCAN_CHANNEL_IDENTIFYING=0x15
var valueAndStatus = pcan.getCanParameter(0x15, value);
var status = valueAndStatus >> 15;
var value = valueAndStatus & 0xff;
if(!status)
{
	scriptThread.messageBox("Critical", "error", ' getCanParameter failed')
}
```

Script timer class

Provides a repetitive timer (QTimer) for generating periodically events. The functions and signals which can be used from script are described in the following chapters.

void start(int msec)

Starts the timer.

Arguments:

- msec: The timer interval in milliseconds

void stop(void)

Stops the timer.

void timeout (void)

This signal is emitted if the timer interval has been elapsed. Scripts can connect a function to this signal.

Timer example

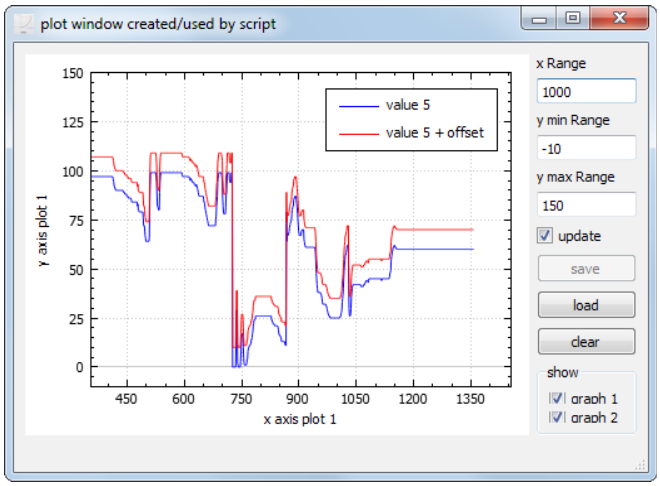
The following Code shows the typically use of the timer class:

```
function updateConsole()
```

```
{
	scriptThread.appendTextToConsole("updateConsole  function called")
}
var consoleTimer = scriptThread.createTimer()
consoleTimer.timeout.connect(updateConsole);
consoleTimer.start(200);
```

Script plot window class

This class provides functions to plot data into a plot window.
This class is derived from the script widget class. Therefore all functions from the script widget class



can be used.
The additional functions and signals which can be used from script are described in the following chapters.

Note: To plot data in an existing script GUI (and not in a separate window) the Script plot widget class can be used.

int addGraphSlot(QString color, QString penStyle, QString name)

This function adds a graph to the diagram.
Arguments:

- color: The color of the graph. Allowed values are: "blue", "red", "yellow", "green" and "black"
- penStyle: The pen style of the graph. Allowed values are: "dash", "dot" and "solid"
- name: The name of the graph

Return: The index of the added graph

void setInitialAxisRangesSlot(double xRange, double yMinValue, double yMaxValue)

Sets the initial ranges of the diagram.
Arguments:

- xRange: The range of the x axis (the x axis starts always with 0)
- yMinValue: The min. values of the y axis
- yMaxValue: The max. value of the y axis

bool addDataToGraphSlot(int graphIndex, double x, double y)

Adds one point to a graph.

Arguments:

- graphIndex: The graph index
- x: The x value of the point
- y: The y value of the point

void setAxisLabels(QString xAxisLabel, QString yAxisLabel)

Sets the axis label.

Arguments:

- xAxisLabel: The label of the x axis
- yAxisLabel: The label of the y axis

void showLegend(bool show)

This function shows or hides the diagram legend.

Arguments:

- show: True=show, false=hide

void clearGraphs(void)

This function clears the data of all graphs.

void removeAllGraphs(void)

This function removes all graphs.

void setMaxDataPointsPerGraph(qint32 maxDataPointsPerGraph)

Sets the max. number of data points per graph (the default is 10.000.000.).

void showHelperElements(bool showXRange, bool showYRange, bool showUpdate, bool showSave, bool showLoad, bool showClear, bool showGraphVisibility, quint32 graphVisibilityMaxSize=80, bool showLegend=true)

Sets the visibility of several plot window elements.

Arguments:

- showXRange: True if the x range input field shall be visible
- showYRange: True if the y range input fields shall be visible
- showUpdate: True if the x update check box shall be visible
- showSave: True if the save button shall be visible
- showLoad: True if the load button shall be visible
- showClear: True if the clear button shall be visible
- showGraphVisibility: True if the show group box
- graphVisibilityMaxSize: The max. width of the show group box
- showLegend: True if the show legend check box shall be visible

void setUpdateInterval(quint32 updateInterval)

Sets the update-interval.

Arguments:

- updateInterval: The new interval

void plotMousePressSignal(double xValue, double yValue, quint32 mouseButton)

Is emitted if the user press a mouse button inside the plot.
Arguments:

- xValue: The x-position of the click
- yValue: The y-position of the click
- mouseButton: The used mouse button (enumeration Qt::MouseButton)

void clearButtonPressedSignal (void)

Is emitted if the user clicks the clear button.

void closedSignal (void)

This signal is emitted if the plot window has been closed. Scripts can connect a function to this signal.

Loading and saving graphs

With the save button all visible graphs can be stored into an image or a comma separated value file (csv). The csv file can be loaded with the load button. The csv file has following format:

```
'name of the graph','color of the graph (rgb)','graph style
(Qt::PenStyle)',x1:y1,x2:y2,...
```

Note: The single graphs are separated with a \n.
Example:

```
graph 1,#0000ff,1,0:0,1:0,2:0,3:0,4:0
graph 2,#00ff00,2,0:0,1:0,2:0,3:0,4:0
```

Plot window example

The following Code shows the typically use of the plot window class:

```
function plotWindowClosedSlot()
{
	scriptThread.stopScript()
}
function clearButtonPressed()
{
	plotWindow.clearGraphs();
}
var plotWindow = scriptThread.createPlotWindow();
plotWindow.setWindowTitle("plot window created/used by script");
plotWindow.setAxisLabels("x axis plot 1", "y axis plot 1");
plotWindow.showLegend(true);
plotWindow.setInitialAxisRanges(100, 0, 15);
var plotWindowGraph1Index = plotWindow.addGraph("blue", "solid", "graph 1");
var plotWindowGraph2Index = plotWindow.addGraph("red", "dot", "graph 2");
plotWindow.showHelperElements(true, true, true, true, true, true);
plotWindow.show();

plotWindow.clearButtonPressedSignal.connect(clearButtonPressed)
plotWindow.closedSignal.connect(plotWindowClosedSlot)

var x = 0;
var y = 0;
for(var i = 0; i < 100; i++)
```

```
{
	x++;
	y++;
	plotWindow.addDataToGraph(plotWindowGraph1Index,  x,  y);
	plotWindow.addDataToGraph(plotWindowGraph2Index,  x + 10,  y);
	if(y > 10)
	{
		y = 0;
	}
}
```

User interface classes

This chapter describes all worker script GUI classes.
The names of all GUI objects (in a worker script) have the following style: UI_"object name in QtDesigner".
Example:
The object name of a button in QtDesigner is MyButton. This button can be accessed by the object name UI_MyButton.
Following (Qt) GUI elements are supported:

- QDialog
- QMainWindow
- QTabWidget
- QToolBox
- QGroupBox
- QLabel
- QAction
- QStatusBar
- QPushButton
- QToolButton
- QCheckBox
- QRadioButton
- QComboBox
- QFontComboBox
- QLineEdit
- QTableWidget
- QListWidget
- QTreeWidget
- QTreeWidgetItem
- QTextEdit
- QProgressBar
- QSlider
- QSpinBox
- QDoubleSpinBox
- QTimeEdit
- QDateEdit
- QDateTimeEdit
- QCalendarWidget
- QSplitter
- QDial

Note: Only the supported GUI elements can be accessed by worker scripts.

Script dialog

A QDialog in the user interface file is passed to the worker script via a ScriptDialog class object. This class is derived from the ScriptWidget class. Therefore all functions from the script widget class can be used.
The additional functions and signals which can be used from script are described in the following chapters.

void finishedSignal(void)

This signal is emitted if the user closes the dialog. Scripts can connect a function to this signal.

Script dialog example

The following Code shows the typically use of the ScriptDialog class:

```
function DialogFinished()
{
	UI_testTextEdit.append("DialogFinished");
	scriptThread.stopScript()
}
UI_Dialog.setWindowTitle("gui example");
Ui_Dialog.setWindowPositionAndSize("100,100,500,500");
UI_Dialog.finishedSignal.connect(DialogFinished);
```

Script main window

A QMainWindow in the user interface file is passed to the script via a ScriptMainWindow class object. This class is derived from the ScriptWidget class. Therefore all functions from the script widget class can be used.
The additional functions and signals which can be used from script are described in the following chapters.

void finishedSignal(void)

This signal is emitted if the user closes the window. Scripts can connect a function to this signal.

Script main window example

The following Code shows the typically use of the ScriptMainWindow class:

```
function MainWindowFinished(e)
{
	scriptThread.stopScript()
}
UI_MainWindow.finishedSignal.connect(MainWindowFinished);
UI_MainWindow.setWindowTitle("main window example");
UI_MainWindow.setWindowPositionAndSize("100,100,300,300");
```

Script tab widget

A QTabWidget in the user interface file is passed to the worker script via a ScriptTab widget class object. This class is derived from the ScriptWidget class. Therefore all functions from the ScriptWidget class can be used.
The additional functions and signals which can be used from script are described in the following chapters.

void setTabText(int index, QString text)

Sets the tab text.

Arguments:

- index: The tab index
- text: The new text

void QString tabText(int index)

Returns the tab text.

Arguments:

- index: The tab index

void setCurrentIndex(int index)

Sets the current tab index.

Arguments:

- index: The new tab index

int currentIndex(void)

Returns the current tab index.

void currentTabChangedSignal(int index)

This signal is emitted if the current tab has been changed.

Scripts can connect a function to this signal.

Script tab widget example

The following Code shows the typically use of the ScriptTabWidget class:

```
//the user has changed the tab index
function CurrentTabChanged(index)
{
	UI_testTextEdit.append("CurrentTabChanged: " + index);
}
UI_tabWidget1.setTabText(0, "new text");
UI_tabWidget1.setCurrentIndex(UI_tabWidget1.currentIndex() + 1);
UI_tabWidget1.currentTabChangedSignal.connect(CurrentTabChanged);
```

Script tool box

A QToolBox in the user interface file is passed to the worker script via a ScriptToolBox class object. This class is derived from the ScriptWidget class. Therefore all functions from the ScriptWidget class can be used.

The functions and signals which can be used from script are described in the following chapters.

void setItemText(int index, QString text)

Sets the item text.

Arguments:

- index: The item index
- text: The new text

void QString itemText(int index)

Returns the item text.

Arguments:

- index: The item index

void setCurrentIndex(int index)

Sets the current item index.

Arguments:

- index: The new item index

int currentIndex(void)

Returns the current item index.

int currentItemChangedSignal(int index)

This signal is emitted if the current item has been changed.

Scripts can connect a function to this signal.

Script tool box example

The following Code shows the typically use of the ScriptToolBox class:

```
//the user has changed the item index
function ToolBoxCurrentItemChanged(index)
{
	UI_testTextEdit.append("ToolBoxCurrentItemChanged: " + index);
}

UI_toolBox.setItemText(1, "new text");
UI_toolBox.setCurrentIndex(UI_toolBox.currentIndex() + 1);
UI_toolBox.currentItemChangedSignal.connect(ToolBoxCurrentItemChanged);
```

Script group box

A QGroupBox in the user interface file is passed to the worker script via a ScriptGroupBox class object. This class is derived from the ScriptWidget class. Therefore all functions from the ScriptWidget class can be used.

The additional functions and signals which can be used from script are described in the following chapters.

void setTitle(QString title)

Sets the group box title.

Arguments:

- title: The new title

QString title(void)

Returns the group box title.

ScriptPlotWidget addPlotWidget(void)

Adds a plot widget to the group box (see chapter Script plot widget for more details).

ScriptCanvas2D addCanvas2DWidget(void)

Adds a plot widget to the group box (see chapter Script Canvas2D for more details).

Script group box example

The following Code shows the typically use of the ScriptGroupBox class:

```
UI_groupBoxProgressAndSlider.setTitle("new title");
```

Script label

A QLabel in the user interface file is passed to the worker script via a ScriptLabel class object.
This class is derived from the ScriptWidget class. Therefore all functions from the ScriptWidget class can be used.
The additional functions and signals which can be used from script are described in the following chapters.

void setText(const QString text)

Sets the label text.
Arguments:

- text: The new text

Example:

```
UI_createProcessProgramLabel.setText("program");
```

QString text(void)

Returns the label text.
Example:

```
var string = UI_createProcessProgramLabel.text();
```

Script action

A QAction in the user interface file is passed to the worker script via a ScriptAction class object.

Important: This class is **not** derived from the ScriptWidget class. Therefore the functions from the ScriptWidget class can not be used.
The functions and signals which can be used from script are described in the following chapters.

void setText(const QString text)

Sets the action text.
Arguments:

- text: The new text

QString text(void)

Returns the action text.

void setChecked(bool checked)

Sets the checked state of the action.
Arguments:

- check: True=checked, false=not checked

bool isChecked(void)
Returns true if the action is checked.

void clickedSignal (void)
This signal is emitted if the user presses the action. Scripts can connect a function to this signal.

Script action example
The following Code shows the typically use of the ScriptAction class:

```
function myActionClicked()
{
	scriptThread.appendTextToConsole("myActionClicked");
}
UI_myAction.setText("new text")
UI_myAction.clickedSignal.connect(myActionClicked);
```

Script status bar
A QStatusBar in the user interface file is passed to the worker script via a ScriptStatusBar class object. This class is derived from the ScriptWidget class. Therefore all functions from the ScriptWidget class can be used.
The additional functions and signals which can be used from script are described in the following chapters.

void showMessage(QString text, int duration)
Shows a message in the status bar.
Arguments:

- text: The message text
- duration: The duration of the message (after this time the status bar will be cleared)

Example:

```
UI_statusbar.showMessage("new message", 1000);
```

Script button
A QPushButton in the user interface file is passed to the worker script via a ScriptButton class object. This class is derived from the ScriptWidget class. Therefore all functions from the ScriptWidget class can be used.
The additional functions and signals which can be used from script are described in the following chapters.

void setText(const QString text)
Sets the button text.
Arguments:

- text: The new text

QString text(void)
Returns the button text.

void setIcon(QString iconFileName)

Sets the icon of the button.

Arguments:

- iconFileName: The path to the icon (absolute path)

void clickedSignal (void)

This signal is emitted if the user presses the button. Scripts can connect a function to this signal.

Script button example

The following Code shows the typically use of the ScriptButton class:

```
function myButtonClicked()
{
	scriptThread.appendTextToConsole("myButtonClicked");
	//disable the button
	UI_myButton.setEnabled(false);
}
UI_myButton.setText("new text")
UI_myButton.setIcon(scriptThread.createAbsolutePath("icons/button.gif")
UI_myButton.clickedSignal.connect(myButtonClicked);
```

Script tool button

A QToolButton in the user interface file is passed to the worker script via a ScriptToolButton class object. This class is derived from the ScriptWidget class. Therefore all functions from the ScriptWidget class can be used.

The additional functions and signals which can be used from script are described in the following chapters.

void setText(const QString text)

Sets the button text.

Arguments:

- text: The new text

QString text(void)

Returns the button text.

void setIcon(QString iconFileName)

Sets the icon of the button.

Arguments:

- iconFileName: The path to the icon (absolute path)

void clickedSignal (void)

This signal is emitted if the user presses the button. Scripts can connect a function to this signal.

Script tool button example

The following Code shows the typically use of the ScriptToolButton class:

```
function myButtonClicked()
{
	scriptThread.appendTextToConsole("myButtonClicked");
```

```
	//disable the button
	UI_myButton.setEnabled(false);
}
UI_myButton.setText("new text")
UI_myButton.setIcon(scriptThread.createAbsolutePath("icons/button.gif")
UI_myButton.clickedSignal.connect(myButtonClicked);
```

Script check box

A QCheckBox in the user interface file is passed to the worker script via a ScriptCheckBox class object. This class is derived from the ScriptWidget class. Therefore all functions from the ScriptWidget class can be used.
The additional functions and signals which can be used from script are described in the following chapters.

void setText(const QString text)

Sets the check box text.
Arguments:

- text: The new text

QString text(void)

Returns the check box text.

void setChecked(bool checked)

Sets the checked state of the check box.
Arguments:

- check: True=checked, false=not checked

bool isChecked(void)

Returns true if the check box is checked.

void clickedSignal (void)

This signal is emitted if the user presses the check box. Scripts can connect a function to this signal.

Script check box example

The following Code shows the typically use of the ScriptCheckBox class:

```
function myCheckBoxClicked()
{
	scriptThread.appendTextToConsole("myCheckBoxClicked");
	//disable the check box
	UI_myCheckBox.setEnabled(false);
}
UI_myCheckBox.setText("new text")
UI_myCheckBox.clickedSignal.connect(myCheckBoxClicked);
```

Script radio button

A QRadioButton in the user interface file is passed to the worker script via a ScriptRadioButton class object. This class is derived from the ScriptWidget class. Therefore all functions from the ScriptWidget class can be used.
The additional functions and signals which can be used from script are described in the following chapters.

Note: If several radio buttons are in the same area, only one radio button can be checked (all other are unchecked automatically).

void setText(const QString text)
Sets the radio button text.
Arguments:

- text: The new text

QString text(void)
Returns the radio button text.

void setChecked(bool checked)
Sets the checked state of the radio button.
Arguments:

- check: True=checked, false=not checked

bool isChecked(void)
Returns true if the radio button is checked.

void clickedSignal (void)
This signal is emitted if the user presses the radio button. Scripts can connect a function to this signal.

Script radio button example
The following Code shows the typically use of the ScriptRadioButton class:

```
function myRadioButtonClicked()
{
	scriptThread.appendTextToConsole("myRadioButtonClicked");
	//disable the check box
	UI_myRadioButton.setEnabled(false);
}
UI_myRadioButton.setText("new text")
UI_myRadioButton.clickedSignal.connect(myRadioButtonClicked);
```

Script combo box and font combo box
A QComboBox and a QFontComboBox in the user interface file is passed to the worker script via a ScriptComboBox class object. This class is derived from the ScriptWidget class. Therefore all functions from the ScriptWidget class can be used.
The additional functions and signals which can be used from script are described in the following chapters.

void addItem(const QString text)
Adds one item to the combo box.
Arguments:

- text: The text of the new item

void insertItem(int index, const QString text)

Inserts one item into the combo box.
Arguments:

- index: The index of the new item
- text: The text of the new item

void removeItem(int index)

Removes one item from the combo box.
Arguments:

- index: The index of the item

void setEditable(bool editable)

Sets the editable property of the combo box. If the editable property is true, then the text of the selected item can be changed.
Arguments:

- editable: True=editable, false=not editable

bool isEditable(void)

Returns true if the combo box is editable. If the editable property is true, then the text of the selected item can be changed.

int currentIndex(void)

Returns the index of the current selected item.

QString currentText(void)

Returns the text of the current selected item.

QString itemText(int index)

Returns the item (identified by index) text.
Arguments:

- index: The index of the item

Return: The item text

void setItemText(int index, const QString text)

Sets the item (identified by index) text.
Arguments:

- index: The index of the item
- text: The new text

void setCurrentText(const QString text)

Sets the text of the current selected item.
Arguments:

- text: The new text

void setCurrentIndex(int index)

Sets the index of the current selected item.
Arguments:

- index: The index of the new selected item

int count(void)

Returns the number of items in the combo box.

void clear(void)

Clears the combo box and removes all items.

void currentTextChangedSignal(QString newText)

This signal is emitted if the text of the current selected item has been changed. Scripts can connect a function to this signal.
Arguments:

- newText: The new text of the changed item

void currentIndexChangedSignal(int currentSelectedIndex)

This signal is emitted if the current selected index has been changed. Scripts can connect a function to this signal.
Arguments:

- currentSelectedIndex: The index of the changed item

Script combo box example

The following Code shows the typically use of the ScriptComboBox class:

```
function myComboBoxCurrentTextChanged(text)
{
	UI_testTextEdit.append("myComboBoxCurrentTextChanged");
}
UI_myComboBox.setCurrentText("myItem");
UI_myComboBox.currentTextChangedSignal.connect(myComboBoxCurrentTextChanged)
```

Script line edit

A QLineEdit in the user interface file is passed to the worker script via a ScriptLineEdit class object. This class is derived from the ScriptWidget class. Therefore all functions from the ScriptWidget class can be used.
The additional functions and signals which can be used from script are described in the following chapters.

void setText(QString text)

Sets the text of the line edit.
Arguments:

- text: The new text

void clear(void)

Clears the line edit.

QString text(void)
Returns the text of the line exit.

bool isReadOnly(void)
Returns true of the line edit is editable.

void setReadOnly(bool readOnly)
Sets the editable property of the line edit.
Arguments:

- readOnly: True=read only, false=editable

void addIntValidator(int bottom, int top)
Adds an int validator to the line edit (this ensures that the line edit contains only integer).
Arguments:

- bottom: The min. value
- top: The max. value

void addDoubleValidator(double bottom, double top, int decimals)
Adds a double validator to the line edit (this ensures that the line edit contains only double values).
Arguments:

- bottom: The min. value
- top: The max. value
- decimals: the max. number of digits after the decimal point

void addRexpExValidator(QString pattern, bool caseSensitiv)
Adds a regular expression validator to the line edit (this ensures that the line edit contains only the allowed values which are specified in the pattern).
Arguments:

- pattern: The pattern (see QRegExp class for more details, http://doc.qt.io/qt-4.8/QRegExp.html)
- caseSensitiv: True if the pattern shall be matched case sensitively

void textChangedSignal(QString currentText)
This signal is emitted if the text of the line edit has been changed. Scripts can connect a function to this signal.
Arguments:

- currentText: The current text of the line edit

Script line edit example
The following Code shows the typically use of the ScriptLineEdit class:

```
function textChangedSlot(text)
{
	UI_testTextEdit.append("textChangedSlot: " + text);
}
UI_lineEdit.addIntValidator(0, 2000);
UI_lineEdit.textChangedSignal.connect(textChangedSlot);
var text = UI_lineEdit.text()
```

Script table widget

A QTableWidget in the user interface file is passed to the worker script via a ScriptTableWidget class object. This class is derived from the ScriptWidget class. Therefore all functions from the ScriptWidget class can be used.
The additional functions and signals which can be used from script are described in the following chapters.

Note:
A selected row in the table can be moved up or down while holding the left mouse button at the row and moving the mouse up and/or down.

QString getText(int row, int column)

Returns the text of one cell.

void setText(int row, int column, QString text)

Sets the text of one cell.
Arguments:

- row: The row of the cell
- column: The column of the cell
- text: The new text

void setVerticalHeaderLabel(int row, QString text)

Sets a vertical header label.
Arguments:

- row: The row of the header
- text: The new text

void setHorizontalHeaderLabel(int column, QString text)

Sets a horizontal header label.
Arguments:

- column: the column of the header
- text: The new text

void setCellEditable(int row, int column, bool editable)

Makes on cell editable or not editable.
Arguments:

- row: The row of the cell
- column: The column of the cell
- editable: True=editable, false=not editable

void setRowCount(int rows)

Sets the row count.
Arguments:

- rows: The new number of rows

void rowCount (void)
Return the row count.

void setColumnCount(int columns)
Sets the column count.
Arguments:

- columns: The new number of columns

int columnCount(void)
Return the column count.

void insertRow(int row)
Inserts one row at row.

void insertColumn(int column)
Inserts one column at column.

void removeRow(int row)
Removes the row at row.

void removeColumn(int column)
Removes the column at column.

void clear(void)
Clears the table (removes all cells).

void setCellBackgroundColor(QString color, int row, int column)
Sets the background color of a single cell.
Arguments:

- color: The color, possible values are: black, white, gray, red, green, blue, cyan, magenta and yellow
- row: The row of the cell
- column: The column of the cell

Example:
```
UI_testSetTextLineEdit.setBackgroundColor("red", 0, 0);
```

void setCellForegroundColor(QString color, int row, int column)
Sets the foreground color of a single cell.
Arguments:

- color: The color, possible values are: black, white, gray, red, green, blue, cyan, magenta and yellow
- row: The row of the cell
- column: The column of the cell

Example:
```
UI_testSetTextLineEdit.setBackgroundColor("red", 0, 0);
```

void sortItems(int column, bool ascendingOrder=true)
Sorts the items in the widget in the specified order(true=AscendingOrder, false=DescendingOrder) by the values in the given column.
Arguments:

- column: The column
- ascendingOrder: true=AscendingOrder, false=DescendingOrder

void resizeColumnToContents(int column)
Resizes the column given to the size of its contents.
Arguments:
- column: The column

void resizeRowToContents(int row)
Resizes the row given to the size of its contents.
Arguments:
- row: The row

void setRowHeight(int row, int height)
Sets the height of the given row to be height.
Arguments:
- row: The row
- height: The height

int rowHeight(int row)
Returns the height of the given row.

void setColumnWidth(int column, int width)
Sets the width of the given column to be width.
Arguments:
- column: The column
- width: The width

int columnWidth(int column)
Returns the width of the given column.

int frameWidth(void)
Returns the width of the frame that is drawn.

int verticalHeaderWidth(void)
Returns the width of the vertical header.

int verticalScrollBarWidth(void)
Returns the width of the vertical scroll bar.

bool isVerticalScrollBarVisible(void)
Returns true if the vertical scroll bar is visible.

bool insertWidget(int row, int column, QString type)
Creates and inserts a script widget into a table cell.

Note: To access the cell widget ScriptWidget* getWidget(int row, int column) must be used.

Arguments:

- row: The row of the cell
- column: The column of the cell
- type: The type of the created widget. Possible type values are:
  - LineEdit
  - ComboBox
  - Button
  - CheckBox
  - SpinBox
  - DoubleSpinBox
  - VerticalSlider
  - HorizontalSlider
  - TimeEdit
  - DateEdit
  - DateTimeEdit
  - CalendarWidget
  - TextEdit
  - Dial

Return: True on success

Example:

```
function TableCheckBoxClicked(checked)
{
	UI_testTextEdit.append("TableCheckBoxClicked: " + checked);
}
UI_testSendTableWidget.insertWidget(3, 1, "CheckBox");
var UI_tableCheckBox1 = UI_testSendTableWidget.getWidget(3, 1);
if(typeof UI_tableCheckBox1 != 'undefined')
{
	UI_tableComboBox1.addItem("val1");
	UI_tableCheckBox1.clickedSignal.connect(TableCheckBoxClicked)
}
```

ScriptWidget getWidget(int row, int column)*

Returns the cell widget. If the cell has no widget then 'invalid' is returned.

Example: see bool insertWidget(int row, int column, QString type).

Note: To determine the class name of the returned ScriptWidget QString getClassName(void) can be used.

void setCellIcon(int row, int column, QString iconFileName)

Sets the icon of a single cell.

Arguments:

- row: The row of the cell
- column: The column of the cell
- iconFileName: The path to the item icon

void rowsCanBeMovedByUser(bool canBeMoved)

If set to true the user can move a selected row up or down (while holding the left mouse button at the row and moving the mouse up and/or down).
Arguments:

- canBeMoved: True for moving selected rows

QVector<ScriptTableCellPosition> getAllSelectedCells(void)

Returns an array which contains the the rows and columns of the selected cells. The array contains one ScriptTableCellPosition object for every selected cell.
The ScriptTableCellPosition object has to attributes which can be accessed: row and column.
Example: see void cellChangedSignal (void).

void cellChangedSignal (void)

This signal is emitted if a cell has been changed. Scripts can connect a function to this signal.
Example:

```
function ReceiveTableSelectionChanged()
{
	var cells = UI_testReceiveTableWidget.getAllSelectedCells();
	for(var i = 0; i < cells.length; i++)
	{
		UI_testTextEdit.append('column: ' + cells[i].column + '  row: '
							+ cells[i].row);
	}

}
UI_testSendTableWidget.cellSelectionChangedSignal.connect(SendTableSelectionChanged)
```

void cellPressedSignal(int row, int column)

This signal is emitted if the user has pressed a cell. Scripts can connect a function to this signal.
Arguments:

- row: The row of the cell
- column: The column of the cell

void cellClickedSignal (int row, int column)

This signal is emitted if the user has clicked a cell. Scripts can connect a function to this signal.
Arguments:

- row: The row of the cell
- column: The column of the cell

void cellDoubleClickedSignal (int row, int column)

This signal is emitted if the user has double clicked a cell. Scripts can connect a function to this signal.
Arguments:

- row: The row of the cell
- column: The column of the cell

void cellChangedSignal (int row, int column)

This signal is emitted if a cell has been changed. Scripts can connect a function to this signal.
Arguments:

- row: The row of the cell
- column: The column of the cell

void horizontalHeaderSectionResizedSignal(int logicalIndex, int oldSize, int newSize)

This signal is emitted if a horizontal header section is resized. Scripts can connect a function to this signal.
Arguments:

- logicalIndex: The section's logical number
- oldSize: The old size
- newSize: The new size

Script table widget example

The following Code shows the typically use of the ScriptTableWidget class:

```
function tableWidgetCellChanged (row, column)
{
       UI_testTextEdit.append("tableWidgetCellChanged: " +
                                  UI_tableWidget.getText(row, column));
}
UI_tableWidget.setRowCount(2);
UI_tableWidget.setColumnCount(2);
UI_tableWidget.setVerticalHeaderLabel(0, "ver1");
UI_tableWidget.setVerticalHeaderLabel(1, "ver2");
UI_tableWidget.setHorizontalHeaderLabel(0, "hor1");
UI_tableWidget.setHorizontalHeaderLabel(1, "hor2");
UI_tableWidget.setCellEditable(0,0, false);
UI_tableWidget.setCellEditable(1,0, false);
UI_tableWidget.insertRow(UI_testSendTableWidget.rowCount() )
UI_tableWidget.setHorizontalHeaderLabel(2, "hor3");
UI_tableWidget.setText(0, 0, "test1");
UI_tableWidget.setText(1, 0, "test2");
UI_tableWidget.setText(2, 0, "test3")
UI_tableWidget.cellChangedSignal.connect(UI_tableWidgetCellChanged);
```

Adjusting the width of the columns so that all columns fit in the complete table

The following code shows an example how to adjust the right column so that all columns fit in the complete table (the table has 2 columns):

```
//Adjust the width of the right column, so that all columns fit in the complete
table.
function adjustTableColmnWidth()
{
       var verticalScrollBarWidth = 0;
       if(UI_TableWidget.isVerticalScrollBarVisible())
       {
       verticalScrollBarWidth = UI_TableWidget.verticalScrollBarWidth();
       }
       UI_TableWidget.setColumnWidth(1, UI_TableWidget.width() -
                                  (UI_TableWidget.columnWidth(0)
                                  + 2 * UI_TableWidget.frameWidth()
                                  + UI_TableWidget.verticalHeaderWidth()
                                  + verticalScrollBarWidth));
}

//The width of a header has been changed.
```

```
function UI_tableHorizontalHeaderSectionResizedSignal(logicalIndex, oldSize,
newSize)
{
	adjustSendTableColmnWidth();
}
//Start a timer which periodically calls adjustTableColmnWidth(if the dialog/table
//size has been changed the columns have to be adjusted)
var tableAdjustWidthTimer = scriptThread.createTimer()
tableAdjustWidthTimer.timeout.connect(adjustTableColmnWidth)
tableAdjustWidthTimer.start(200);

UI_testSendTableWidget.horizontalHeaderSectionResizedSignal.connect(UI_tableHorizont
alHeaderSectionResizedSignal)
```

Script list widget

A QListWidget in the user interface file is passed to the worker script via a ScriptListWidget class object. This class is derived from the ScriptWidget class. Therefore all functions from the ScriptWidget class can be used.
The additional functions and signals which can be used from script are described in the following chapters.

Note:
A selected row in the table can be moved up or down while holding the left mouse button at the row and moving the mouse up and/or down.

void insertNewItem (int row, QString itemText, QString iconFileName)

Inserts a new list item.
Arguments:

- row: The row of the new item
- itemText: The text of the new item
- iconFileName: The path to the item icon (if empty then no icon will be added)

int rowCount(void)

Returns the number of rows in the list widget.

void removeItem(int row)

Removes a item from the list widget.
Arguments:

- row: The row of the item

int currentSelectedRow(void)

Returns the current selected row.

void setCurrentRow(int row)

Sets the current selected row.
Arguments:

- row: The row of the item

void clear(void)
Clears the list widget.

void setItemBackgroundColor(int row, QString color)
Sets the background color of an item.
Arguments:

- row: The row of the item
- color: The color, possible values are: black, white, gray, red, green, blue, cyan, magenta and yellow

void setItemForegroundColor(int row, QString color)
Sets the foreground color of an item.
Arguments:

- row: The row of the item
- color: The color, possible values are: black, white, gray, red, green, blue, cyan, magenta and yellow

QString getItemText(int row)
Returns the item text.
Arguments:

- row: The row of the item

void setItemText(int row, QString text)
Sets the item text.
Arguments:

- row: The row of the item
- text: The new text

void setItemIcon(int row, QString iconFileName)
Sets the item icon.
Arguments:

- row: The row of the item
- iconFileName: The path to the item icon

void sortItems(bool ascendingOrder=true)
Sorts the items in the widget in the specified order(true=AscendingOrder, false=DescendingOrder).

void currentRowChangedSignal(int currentRow)
This signal is emitted if the current row selection has been changed.
Arguments:

- currentRow: The index of the current selected item

void itemClickedSignal(int row)
This signal is emitted if a row has been clicked.
Arguments:

- row: The row of the item

void itemDoubleClickedSignal(int row)

This signal is emitted if a row has been double clicked.

Arguments:

- row: The row of the item

Script list widget example

The following Code shows the typically use of the ScriptListWidget class:

```
function UI_listWidgetCurrentRowChanged(currentRow)
{
	UI_testTextEdit.append("UI_listWidgetCurrentRowChanged: " + currentRow);
}
function UI_listWidgetItemClicked(currentRow)
{
	UI_testTextEdit.append("UI_listWidgetItemClicked: " + currentRow);
}
function UI_listWidgetItemDoubleClicked(currentRow)
{
	UI_testTextEdit.append("UI_listWidgetItemDoubleClicked: " + currentRow);
}

UI_sendListWidget.insertNewItem(UI_sendListWidget.rowCount(), "item0",
scriptThread.createAbsolutePath("icons/folder.gif"));
UI_sendListWidget.seItemBackgroundColor(UI_sendListWidget.rowCount() - 1, "red")

UI_sendListWidget.insertNewItem(UI_sendListWidget.rowCount(), "item1",
scriptThread.createAbsolutePath("icons/browser.ico"));
UI_sendListWidget.setCurrentRow(0);

UI_sendListWidget.currentRowChangedSignal.connect(UI_listWidgetCurrentRowChanged);
UI_sendListWidget.itemClickedSignal.connect(UI_listWidgetItemClick)
UI_sendListWidget.itemDoubleClickedSignal.connect(UI_listWidgetItemDoubleClicked);
```

Script tree widget

A QTreeWidget in the user interface file is passed to the worker script via a ScriptTreeWidget class object. This class is derived from the ScriptWidget class. Therefore all functions from the ScriptWidget class can be used.

The additional functions and signals which can be used from script are described in the following chapters.

ScriptTreeWidgetItem createScriptTreeWidgetItem(void)*

Creates a script tree widget item (for more details about the script tree widget item see chapter Script tree widget item)

Note: A created ScriptTreeWidgetItem has to be inserted in a ScriptTreeWidget or has to be deleted with ScriptTreeWidgetItem::deleteItem() (memory leak).

void setHeaderLabels (QStringList labels)

Adds a column in the header for each item in the labels list, and sets the label for each column.

Note that setHeaderLabels() won't remove existing columns.

Example:

```
var headerList = Array("header 1", "header 2");
```

```
UI_treeWidget.setHeaderLabels(headerList);
```

void setColumnWidth(int column, int size)
Sets the width of a column.

int getColumnWidth(int column)
Returns the width of a column.

void addTopLevelItem (ScriptTreeWidgetItem item)*
Appends the item as a top-level item in the widget.

void insertTopLevelItem (int index, ScriptTreeWidgetItem item)*
Inserts the item at index in the top level in the view.
If the item has already been inserted somewhere else it won't be inserted.

int topLevelItemCount(void)
Returns the number of top level items.

ScriptTreeWidgetItem invisibleRootItem(void)*
Returns the tree widget's invisible root item.
The invisible root item provides access to the tree widget's top-level items through the ScriptTreeWidgetItem API, making it possible to write functions that can treat top-level items and their children in a uniform way; for example, recursive functions.

ScriptTreeWidgetItem itemAbove(ScriptTreeWidgetItem* item)*
Returns the item above the given item. If to item is above then it returns null.

ScriptTreeWidgetItem itemBelow(ScriptTreeWidgetItem* item)*
Returns the item below the given item. If to item is above then it returns null.

ScriptTreeWidgetItem takeTopLevelItem(int index)*
Removes the top-level item at the given index in the tree and returns it, otherwise returns null.
Note: A removed ScriptTreeWidgetItem has to be inserted in a ScriptTreeWidget or has to be deleted with ScriptTreeWidgetItem::deleteItem() (memory leak).

ScriptTreeWidgetItem topLevelItem(int index)*
Returns the top level item at the given index, or null if the item does not exist.

void resizeColumnToContents(int column)
Resizes the column given to the size of its contents.

int columnCount(void)
Returns the number of columns displayed in the tree widget.

void setColumnCount(int columns)
Sets the number of columns displayed in the tree widget.

void expandItem(ScriptTreeWidgetItem item)*

Expands the item. This causes the tree containing the item's children to be expanded.

void expandAll(void)

Expands all expandable items.

void setCurrentItem (ScriptTreeWidgetItem item)*

Sets the current item in the tree widget.

ScriptTreeWidgetItem currentItem(void)*

Returns current item in the tree widget.

void sortItems(int column, bool ascendingOrder=true)

Sorts the items in the widget in the specified order(true=AscendingOrder, false=DescendingOrder) by the values in the given column.

*void itemClickedSignal(ScriptTreeWidgetItem *item, int column)*

This signal is emitted if an item has been clicked.

*void itemDoubleClickedSignal(ScriptTreeWidgetItem *item, int column)*

This signal is emitted if an item has been double clicked.

*void currentItemChangedSignal(ScriptTreeWidgetItem *current, ScriptTreeWidgetItem *previous)*

This signal is emitted if the current item changes. The current item is specified by current, and this replaces the previous current item.

Script tree widget example

The following Code shows the typically use of the ScriptTreeWidget class:

```
function UI_treeWidgetItemClicked(item, column)
{
	UI_testTextEdit.append("UI_treeWidgetItemClicked: " + item.text(column));
}

function UI_treeWidgetItemDoubleClicked(item, column)
{
	UI_testTextEdit.append("UI_treeWidgetItemDoubleClicked: " +
				item.text(column));
}

UI_treeWidget.itemClickedSignal.connect(UI_treeWidgetItemClicked);
UI_treeWidget.itemDoubleClickedSignal.connect(UI_treeWidgetItemDoubleClicked);

UI_treeWidget.setColumnWidth(0, 100);

var treeItem1 = UI_treeWidget.createScriptTreeWidgetItem();
treeItem1.setText(0, "value1");
treeItem1.setData(0, 0, "value 1");
```

```
treeItem1.setText(0, treeItem1.data(0, 0));
treeItem1.setItemIcon(0, scriptThread.createAbsolutePath("icons/openfolder.gif"));
treeItem1.setText(1, "value2");
treeItem1.setItemIcon(1, scriptThread.createAbsolutePath("icons/folder.gif"));
UI_treeWidget.addTopLevelItem(treeItem1);

var treeItem2 = UI_treeWidget.createScriptTreeWidgetItem();
treeItem2.setText(0, "value3");
treeItem2.setItemIcon(0, scriptThread.createAbsolutePath("icons/browser.ico"));
treeItem2.setText(1, "value4");
treeItem2.setItemIcon(1, scriptThread.createAbsolutePath("icons/browser.ico"));
UI_treeWidget.insertTopLevelItem(UI_treeWidget.topLevelItemCount(), treeItem2)
treeItem2.setBackgroundColor(0, "red");
treeItem2.setForegroundColor(1, "blue");
treeItem1.addChild(treeItem2);

UI_treeWidget.expandItem(treeItem1);
```

Script tree widget item

A QTreeWidgetItem (one element in a QTreeWidget object) in the user interface file is passed to the worker script via a ScriptTreeWidgetItem class object.

Important: This class is **not** derived from the ScriptWidget class. Therefore the functions from the ScriptWidget class can not be used.

Note: A created ScriptTreeWidgetItem has to be inserted in a ScriptTreeWidget or has to be deleted with ScriptTreeWidgetItem::deleteItem() (memory leak).
The functions and signals which can be used from script are described in the following chapters.

void setText(int column, QString text)

Sets the text to be displayed in the given column to the given text.

QString text(int column)

Returns the text in the specified column.

void setItemIcon(int column, QString iconFileName)

Sets the item icon.

void addChild(ScriptTreeWidgetItem child)*

Appends the child item to the list of children.

int childCount(void)

Returns the number of child items.

void insertChild (int index, ScriptTreeWidgetItem child)*

Inserts the child item at index in the list of children.
If the child has already been inserted somewhere else it won't be inserted again.

ScriptTreeWidgetItem takeChild (int index)*

Removes the item at index and returns it, otherwise return null.
Note: A temoved ScriptTreeWidgetItem has to be inserted in a ScriptTreeWidget or has to be deleted with ScriptTreeWidgetItem::deleteItem() (memory leak).

void deleteItem(void)

Deletes the current item.

int indexOfChild(ScriptTreeWidgetItem child)*

Returns the index of the given child in the item's list of children.

void sortChildren(int column, bool ascendingOrder)

Sorts the children of the item using the given order(true=AscendingOrder, false=DescendingOrder) by the values in the given column.

ScriptTreeWidgetItem parent (void)*

Returns the item's parent.

int columnCount(void)

Returns the number of columns in the item.

void setBackgroundColor(int column, QString color)

Sets the background color of the label in the given column to the specified color.
Possible colors are: black, white, gray, red, green, blue, cyan, magenta and yellow.

void setForegroundColor(int column, QString color)

Sets the foreground color of the label in the given column to the specified color.
Possible colors are: black, white, gray, red, green, blue, cyan, magenta and yellow.

bool isExpanded(void)

Returns true if the item is expanded, otherwise returns false.

void setExpanded(bool expand)

Expands the item if expand is true, otherwise collapses the item.

void setData (int column, quint8 role, QString value)

Sets the value for the item's column and role to the given value.
The role describes the type of data specified by value.

QString data(int column, quint8 role)

Returns the value for the item's column and role to the given value.
The role describes the type of data specified by value.

void setDisabled(bool disabled)

Disables the item if disabled is true; otherwise enables the item.

bool isDisabled(void)
Returns true if the item is disabled; otherwise returns false.

Script tree widget item example
The typically use of the ScriptTreeWidgetItem class is shown in chapter Script tree widget example

Script text edit
A QTextEdit in the user interface file is passed to the worker script via a ScriptTextEdit class object. This class is derived from the ScriptWidget class. Therefore all functions from the ScriptWidget class can be used.
The additional functions and signals which can be used from script are described in the following chapters.

int verticalScrollBarValue(void)
Returns the vertical scroll bar value.

void verticalScrollBarSetValue(int value)
Sets the vertical scroll bar value.
Arguments:

- value: The new value

QString toPlainText(void)
Returns the content of the text edit as plain text.

QString toHtml(void)
Returns the content of the text edit as html.

void setMaxChars(int maxChars)
Sets the maximum number of characters in the text edit.
Arguments:

- maxChars: The new value for max. characters

QString replaceNonHtmlChars(QString text)
Replaces the characters '\n',' ', '<' and '>' to their html representation.
Arguments:

- text: The old text

Return: The converted text

void moveTextPositionToEnd(void)
Moves the curser to the end of the text edit.

void setFontPointSize(qreal fontSize)
Sets the font size.
Arguments:

- fontSize: The new font size

void setFontFamily (qreal fontFamily)
Sets the font family.

Arguments:

- fontFamily: The new font family

void clear(void)

Clears the text edit.

void insertPlainText(const QString text, bool atTheEnd=true)

Inserts plain text into the text edit.

Arguments:

- text: The text
- atTheEnd: If true then the text is inserted at the end if false then the text is inserted at the cursor position

void insertHtml(QString htmlString, bool atTheEnd=true)

Inserts HTML text into the text edit.

Arguments:

- htmlString: The HTML string
- atTheEnd: If true then the text is inserted at the end if false then the text is inserted at the cursor position

void append(QString text)

Appends text at the end of text edit (includes a new line) and moves the cursor to the end of the text.

Arguments:

- text: The text

void setPlainText(QString text)

Sets the text of the text edit (plain text).

Arguments:

- text: The text

void setText(QString text)

Sets the text of the text edit.

Arguments:

- text: The text

void lockScrolling(bool lock)

Locks or unlocks the scrolling of the vertical scroll bar.

Arguments:

- lock: true for locking and false for unlocking

void textChangedSignal (void)

This signal is emitted if the text of the text edit has been changed. Scripts can connect a function to this signal.

Script progress bar

A QProgressBar in the user interface file is passed to the script via a ScriptProgressBar class object.

This class is derived from the ScriptWidget class. Therefore all functions from the ScriptWidget class can be used.
The additional functions and signals which can be used from script are described in the following chapters.

void reset(void)
Resets the progress bar. The progress bar rewinds and shows no progress.

void setRange(int minimum, int maximum)
Sets the progress bar's minimum and maximum values to minimum and maximum respectively.
Arguments:

- minimum: The minimum
- maximum: The maximum

void setMinimum(int minimum)
Sets the progress bar's minimum value.
Arguments:

- minimum: The minimum

void setMaximum(int maximum)
Sets the progress bar's maximum value.
Arguments:

- maximum: The maximum

void setValue(int value)
Sets the progress bar's current value.
Arguments:

- value: The current value

Script progress bar example
The following Code shows the typically use of the ScriptProgressBar class:

```
UI_progressBar.setMinimum(0);
UI_progressBar.setMaximum(100);
UI_progressBar.setValue(10);
```

Script slider
A QSlider (horizontal and vertical) in the user interface file is passed to the script via a ScriptSlider class object. This class is derived from the ScriptWidget class. Therefore all functions from the ScriptWidget class can be used.
The additional functions and signals which can be used from script are described in the following chapters.

void setRange(int min, int max)
Sets the slider's minimum to min and its maximum to max.
Arguments:

- min: The minimum
- max: The maximum

void setValue(int value)
Sets the slider's current value.
Arguments:
- value: The current value

int value(void)
Returns the slider's current value.

void valueChangedSignal(int value)
This signal is emitted if the value of the slider has been changed.

Script slider example
The following Code shows the typically use of the ScriptSlider class:

```
function UI_horizontalSliderValueChanged(value)
{
	UI_testTextEdit.append("UI_horizontalSliderValueChanged");
}
UI_horizontalSlider.valueChangedSignal.connect(UI_horizontalSliderValueChanged);
UI_horizontalSlider.setRange(0,100);
UI_horizontalSlider.setValue(0);
```

Script spin box
A QSpinBox in the user interface file is passed to the worker script via a ScriptSpinBox class object. This class is derived from the ScriptWidget class. Therefore all functions from the ScriptWidget class can be used.
The additional functions and signals which can be used from script are described in the following chapters.

void setRange(int min, int max)
Sets the spin box's minimum to min and its maximum to max.
Arguments:
- min: The minimum
- max: The maximum

void setValue(int value)
Sets the spin box's current value.
Arguments:
- value: The current value

int value(void)
Returns the spin box's current value.

void setSingleStep(int value)
If the user uses the arrows to change the spin box's value the value will be

incremented/decremented by the amount of the single step. The default value is 1.
Setting a single step value of less than 0 does nothing.
Arguments:

- value: The new value

int singleStep(void)

Returns the single step value.

void valueChangedSignal(int value)

This signal is emitted if the value of the spin box has been changed.

Script spin box example

The following Code shows the typically use of the ScriptSpinBox class:

```
function UI_spinBoxValueChanged(value)
{
	UI_testTextEdit.append("UI_spinBoxValueChanged: " + value);
}
UI_spinBox.valueChangedSignal.connect(UI_spinBoxValueChanged);
UI_spinBox.setRange(0,100);
UI_spinBox.setValue(0);
```

Script double spin box

A QDoubleSpinBox in the user interface file is passed to the script via a ScriptDoubleSpinBox class object. This class is derived from the ScriptWidget class. Therefore all functions from the ScriptWidget class can be used.
The additional functions and signals which can be used from script are described in the following chapters.

void setRange(double min, double max)

Sets the spin box's minimum to min and its maximum to max.
Arguments:

- min: The minimum
- max: The maximum

void setValue(double value)

Sets the spin box's current value.
Arguments:

- value: The current value

double value(void)

Returns the spin box's current value.

void setSingleStep(double value)

If the user uses the arrows to change the spin box's value the value will be incremented/decremented by the amount of the single step. The default value is 1.0.
Setting a single step value of less than 0 does nothing.

Arguments:

- value: The new value

double singleStep(void)

Returns the single step value.

void setDecimals(int value)

Sets the precision of the spin box, in decimals.
Arguments:

- value: The new value

int decimals(void)

Returns the precision of the spin box, in decimals.

void valueChangedSignal(double value)

This signal is emitted if the value of the spin box has been changed.

Script double spin box example

The following Code shows the typically use of the ScriptDoubleSpinBox class:

```
function UI_doubleSpinBoxValueChanged(value)
{
	UI_testTextEdit.append("UI_doubleSpinBoxValueChanged: " + value);
}
UI_doubleSpinBox.valueChangedSignal.connect(UI_doubleSpinBoxValueChanged);
UI_doubleSpinBox.setRange(0,100);
UI_doubleSpinBox.setDecimals(2);
UI_doubleSpinBox.setValue(0);
```

Script time edit

A QTimeEdit in the user interface file is passed to the worker script via a ScriptTimeEdit class object. This class is derived from the ScriptWidget class. Therefore all functions from the ScriptWidget class can be used.
The additional functions and signals which can be used from script are described in the following chapters.

void setTime(QString time)

Sets the time.
Arguments:

- time: The time (must satisfy the display format)

QString getTime(void)

Returns the time.

void setDisplayFormat(QString format)

Sets the display format.
Arguments:

- format: The new display format (see chapter Time format)

QString getDisplayFormat(void)
Returns the display format.

void timeChangedSignal(QString time)
This signal is emitted if the value of the time edit has been changed.

Time format
Following expressions may be used for the time format:

Expression	Output
h	the hour without a leading zero (0 to 23 or 1 to 12 if AM/PM display)
hh	the hour with a leading zero (00 to 23 or 01 to 12 if AM/PM display)
m	the minute without a leading zero (0 to 59)
mm	the minute with a leading zero (00 to 59)
s	the second without a leading zero (0 to 59)
ss	the second with a leading zero (00 to 59)
z	the milliseconds without leading zeroes (0 to 999)
zzz	the milliseconds with leading zeroes (000 to 999)
AP	interpret as an AM/PM time. AP must be either "AM" or "PM".
ap	Interpret as an AM/PM time. ap must be either "am" or "pm".

All other input characters will be treated as text. Any sequence of characters that are enclosed in single quotes will also be treated as text and not be used as an expression.

```
//time is 12:01.00
UI_timeEdit.setDisplayFormat("m'mm'hcarss");
UI_timeEdit.setTime("1mm12car00");
```

Expressions that do not expect leading zeroes to be given (h, m, s and z) are greedy.
This means that they will use two digits even if this puts them outside the range of accepted values and leaves too few digits for other sections.
For example, the following string could have meant 00:07:10, but the m will grab two digits, resulting in an invalid time:

```
//invalid
UI_timeEdit.setDisplayFormat("hh:ms");
UI_timeEdit.setTime("00:710");
```

Any field that is not represented in the format will be set to zero. For example:

```
//time is 00:01:30.000
UI_timeEdit.setDisplayFormat("m.s");
UI_timeEdit.setTime("1.30");
```

Script time edit example
The following Code shows the typically use of the ScriptTimeEdit class:

```
function UI_timeEditTimeChanged(time)
```

```
{
	UI_testTextEdit.append("UI_timeEditTimeChanged: " + time);
}
UI_timeEdit.setDisplayFormat("hh:mm:ss");
UI_timeEdit.setTime("10:09:08");
UI_timeEdit.timeChangedSignal.connect(UI_timeEditTimeChanged);
```

Script date edit

A QDateEdit in the user interface file is passed to the worker script via a ScriptDateEdit class object. This class is derived from the ScriptWidget class. Therefore all functions from the ScriptWidget class can be used.
The additional functions and signals which can be used from script are described in the following chapters.

void setDate(QString date)

Sets the date.
Arguments:

- date: The date (must satisfy the display format)

QString getDate(void)

Returns the date.

void setDisplayFormat(QString format)

Sets the display format.
Arguments:

- format: The new display format (see chapter Date format)

QString getDisplayFormat(void)

Returns the display format.

void dateChangedSignal(QString date)

This signal is emitted if the value of the date edit has been changed.

Date format

Following expressions may be used for the date format:

Expression	**Output**
d	The day as a number without a leading zero (1 to 31)
dd	The day as a number with a leading zero (01 to 31)
ddd	The abbreviated localized day name (e.g. 'Mon' to 'Sun'). Uses the system locale to localize the name, i.e. QLocale::system().
dddd	The long localized day name (e.g. 'Monday' to 'Sunday'). Uses the system locale to localize the name, i.e. QLocale::system().
M	The month as a number without a leading zero (1 to 12)
MM	The month as a number with a leading zero (01 to 12)
MMM	The abbreviated localized month name (e.g. 'Jan' to 'Dec'). Uses the system locale to localize

	the name, i.e. QLocale::system().
MMMM	The long localized month name (e.g. 'January' to 'December'). Uses the system locale to localize the name, i.e. QLocale::system().
yy	The year as two digit number (00 to 99)
yyyy	The year as four digit number. If the year is negative, a minus sign is prepended in addition.

All other input characters will be treated as text. Any sequence of characters that are enclosed in single quotes will also be treated as text and will not be used as an expression. For example:

```
//date is 1 December 2003
UI_dateEdit.setDisplayFormat("d'MM'MMcaryyyy");
UI_dateEdit.setDate("1MM12car2003");
```

The expressions that don't expect leading zeroes (d, M) will be greedy. This means that they will use two digits even if this will put them outside the accepted range of values and leaves too few digits for other sections. For example, the following format string could have meant January 30 but the M will grab two digits, resulting in an invalid date:

```
//invalid
UI_dateEdit.setDisplayFormat("Md");
UI_dateEdit.setDate("130");
```

For any field that is not represented in the format the following defaults are used:

Field	Default value
Year	1900
Month	1
Day	1

The following examples demonstrate the default values:

```
//January 30 1900
UI_dateEdit.setDisplayFormat("M.d");
UI_dateEdit.setDate("1.30");

//January 10, 2000
UI_dateEdit.setDisplayFormat("yyyyMMdd");
UI_dateEdit.setDate("20000110");

//January 10, 2000
UI_dateEdit.setDisplayFormat("yyyyMd");
UI_dateEdit.setDate("20000110");
```

Script date edit example

The following Code shows the typically use of the ScriptDateEdit class:

```
function UI_dateEditDateChanged(date)
{
	UI_testTextEdit.append("UI_dateEditDateChanged: " + date);
}
UI_dateEdit.setDisplayFormat("dd.MM.yyyy");
UI_dateEdit.setDate("01.02.2014");
UI_dateEdit.dateChangedSignal.connect(UI_dateEditDateChanged);
```

Script date time edit

A QDateTimeEdit in the user interface file is passed to the worker script via a ScriptDateTimeEdit class object. This class is derived from the ScriptWidget class. Therefore all functions from the ScriptWidget class can be used.
The additional functions and signals which can be used from script are described in the following chapters.

void setDateTime(QString dateTimeString)

Sets the date and the time.
Arguments:

- date: The date and time (must satisfy the display format)

QString getDateTime(void)

Returns the date and time.

void setDisplayFormat(QString format)

Sets the display format.
Arguments:

- format: The new display format (see chapter Date format and Time format)

Example:

```
UI_dateTimeEdit.setDisplayFormat("dd.MM.yyyy hh:mm:ss");
```

QString getDisplayFormat(void)

Returns the display format.

void dateTimeChangedSignal(QString date)

This signal is emitted if the value of the date time edit has been changed.

Script date time edit example

The following Code shows the typically use of the ScriptDateTimeEdit class:

```
//the user has changed the date and/or the time
function UI_dateTimeEditTimeChanged(dateTime)
{
	UI_testTextEdit.append("UI_dateTimeEditTimeChanged: " + dateTime);
}
UI_dateTimeEdit.setDisplayFormat("dd.MM.yyyy hh:mm:ss");
UI_dateTimeEdit.setDateTime("01.01.2016 10:09:08");
UI_dateTimeEdit.dateTimeChangedSignal.connect(UI_dateTimeEditTimeChanged);
```

Script calendar widget

A QCalendarWidget in the user interface file is passed to the worker script via a ScriptCalendarWidget class object. This class is derived from the ScriptWidget class. Therefore all functions from the ScriptWidget class can be used.
The additional functions and signals which can be used from script are described in the following chapters.

void setSelectedDate(QString dateString)

Sets the selected date.

Arguments:

- date: The date (must satisfy the date format)

QString getSelectedDate(void)

Returns the selected date.

void setDateFormat(QString format)

Sets the date format.

Arguments:

- format: The new date format (see chapter Date format)

QString getDateFormat(void)

Returns the date format.

void setDateRange(QString min, QString max)

Sets the minimum and the maximum date.

Arguments:

- min: the min. date (must satisfy the date format)
- max: the max. date (must satisfy the date format)

void selectionChangedSignal(QString date)

This signal is emitted if the selected date has been changed.

Script calendar widget example

The following Code shows the typically use of the ScriptCalendarWidget class:

```
//the user has changed the date
function UI_calendarSelectionChangedSignal(date)
{
	UI_testTextEdit.append("UI_calendarSelectionChangedSignal: " + date);
}
UI_calendar.setDateFormat("dd.MM.yyyy");
UI_calendar.setDateRange("01.01.2014", "01.01.2016");
UI_calendar.setSelectedDate("01.02.2015");
UI_calendar.selectionChangedSignal.connect(UI_calendarSelectionChangedSignal);
```

Script splitter

A QSplitter in the user interface file is passed to the worker script via a ScriptSplitter class object. This class is derived from the ScriptWidget class. Therefore all functions from the ScriptWidget class can be used.

The additional functions and signals which can be used from script are described in the following chapters.

QList<int> sizes(void)

Returns a list of the size parameters of all the widgets in this splitter.

If the splitter's orientation is horizontal, the list contains the widgets width in pixels,

from left to right; if the orientation is vertical, the list contains the widgets height in pixels, from top to bottom.

void setSizes (QList<int> list)

Sets the child widgets respective sizes to the values given in the list.
If the splitter is horizontal, the values set the widths of each widget in pixels, from left to right.
If the splitter is vertical, the heights of each widget is set, from top to bottom.
Extra values in the list are ignored. If list contains too few values, the result is undefined but the program will still be well-behaved.
The overall size of the splitter widget is not affected. Instead, any additional/missing space is distributed amongst the widgets according to the relative weight of the sizes.
If you specify a size of 0, the widget will be invisible. The size policies of the widgets are preserved. That is, a value smaller then the minimal size hint of the respective widget will be replaced by the value of the hint.

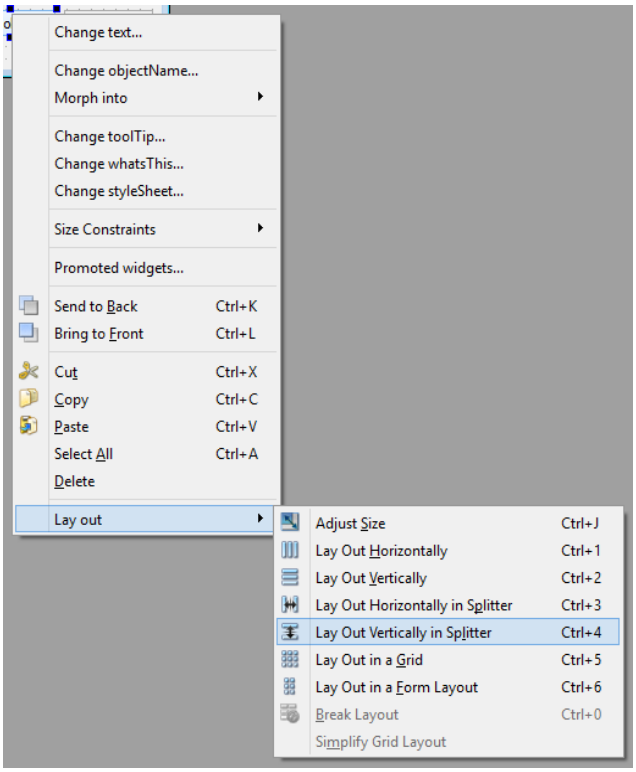
Script splitter example

The following Code shows the typically use of the script splitter class:

```
var sizeList = UI_SecondDialogSplitter.sizes();
//Change the sizes.
sizeList[0] -= 50;
sizeList[1] += 50;
UI_SecondDialogSplitter.setSizes(sizeList);
```

Note: To add a splitter in the designer following must be done:

- select several items
- press the right mouse button
- go to the 'lay out' menu
- select 'lay out vertically in splitter' or 'Lay out horizontally in splitter' (see below)



Script dial

A QDial in the user interface file is passed to the worker script via a ScriptDial class object.
This class is derived from the ScriptWidget class. Therefore all functions from the ScriptWidget class can be used.
The additional functions and signals which can be used from script are described in the following chapters.

void setRange(int min, int max)

Sets the dial's minimum to min and its maximum to max.
Arguments:

- min: The minimum
- max: The maximum

void setValue(int value)

Sets the dial's current value.
Arguments:

- value: The current value

int value(void)

Returns the dial's current value.

void valueChangedSignal(int value)

This signal is emitted if the value of the dial has been changed.

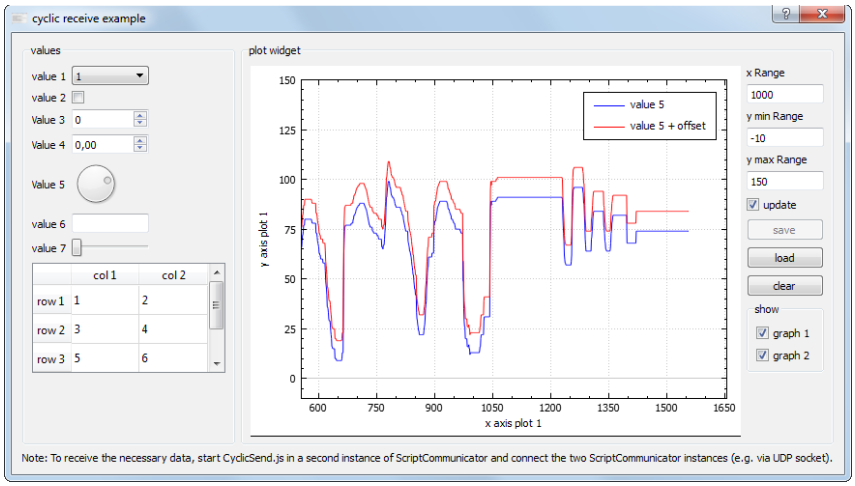
Script dial example

The following Code shows the typically use of the ScriptDial class:

```
function UI_dialValueChanged(value)
{
	UI_testTextEdit.append("UI_dialValueChanged");
}
UI_dial.valueChangedSignal.connect(UI_dialValueChanged);
UI_dial.setRange(0,100);
UI_dial.setValue(0);
```

Script plot widget

This class provides functions to plot data into a script GUI.



If you want a separate window for plotting data, then use a plot window instead (see chapter Script plot window class).

To create a ScriptPlotWidget following must be done:

- add a group box to the script GUI (QtDesigner)
- call ScriptGroupBox::addPlotWidget (see chapter ScriptPlotWidget addPlotWidget(void))

Important: This class is **not** derived from the ScriptWidget class. Therefore the functions from the ScriptWidget class can not be used.

The functions and signals which can be used from script are described in the following chapters.

int addGraphSlot(QString color, QString penStyle, QString name)

This function adds a graph to the diagram.

Arguments:

- color: The color of the graph. Allowed values are: "blue", "red", "yellow", "green" and "black"
- penStyle: The pen style of the graph. Allowed values are: "dash", "dot" and "solid"
- name: The name of the graph

Return: The index of the added graph

void setInitialAxisRangesSlot(double xRange, double yMinValue, double yMaxValue)

Sets the initial ranges of the diagram.

Arguments:

- xRange: The range of the x axis (the x axis starts always with 0)
- yMinValue: The min. values of the y axis
- yMaxValue: The max. value of the y axis

bool addDataToGraphSlot(int graphIndex, double x, double y)

Adds one point to a graph.
Arguments:

- graphIndex: The graph index
- x: The x value of the point
- y: The y value of the point

void setAxisLabels(QString xAxisLabel, QString yAxisLabel)

Sets the axis label.
Arguments:

- xAxisLabel: The label of the x axis
- yAxisLabel: The label of the y axis

void showLegend(bool show)

This function shows or hides the diagram legend.
Arguments:

- show: True=show, false=hide

void clearGraphs(void)

This function clears the data of all graphs.

void removeAllGraphs(void)

This function removes all graphs.

void showHelperElements(bool showXRange, bool showYRange, bool showUpdate, bool showSave, bool showLoad, bool showClear, bool showGraphVisibility, quint32 graphVisibilityMaxSize=80, bool showLegend=true)

Sets the visibility of several plot widget elements.
Arguments:

- showXRange: True if the x range input field shall be visible
- showYRange: True if the y range input fields shall be visible
- showUpdate: True if the x update check box shall be visible
- showSave: True if the save button shall be visible
- showLoad: True if the load button shall be visible
- showClear: True if the clear button shall be visible
- showGraphVisibility: True if the show group box
- graphVisibilityMaxSize: The max. width of the show group box
- showLegend: True if the show legend check box shall be visible

void setMaxDataPointsPerGraph(qint32 maxDataPointsPerGraph)

Sets the max. number of data points per graph (the default is 10.000.000.).

void setUpdateInterval(quint32 updateInterval)

Sets the update-interval.

Arguments:

- updateInterval: The new interval

void clearButtonPressedSignal (void)

Is emitted if the user clicks the clear button.

void plotMousePressSignal(double xValue, double yValue, quint32 mouseButton)

Is emitted if the user press a mouse button inside the plot.

Arguments:

- xValue: The x-position of the click
- yValue: The y-position of the click
- mouseButton: The used mouse button (enumeration Qt::MouseButton)

Loading and saving graphs

With the save button all visible graphs can be stored into an image or a comma separated value file (csv). The csv file can be loaded with the load button. For more details see chapter Loading and saving graphs.

Script plot widget example

The following Code shows the typically use of the ScriptPlotWidget class:

```
//Is called if the user clicks the button.
function clearButtonPressed()
{
	plotXCounter = 0;
	plotWindow.clearGraphs();
	plotWidget.clearGraphs();
}

var plotWidget = UI_PlotGroupBox.addPlotWidget();
plotWidget.setAxisLabels("x axis plot 1", "y axis plot 1");
plotWidget.showLegend(true);
plotWidget.setInitialAxisRanges(100, 0, 15);
var plotWidgetGraph1Index = plotWidget.addGraph("blue", "solid", "value 5");
var plotWidgetGraph2Index = plotWidget.addGraph("red", "solid", "value 5 + offset");
plotWidget.showHelperElements(true, true, true, true, true, true);
plotWidget.clearButtonPressedSignal.connect(clearButtonPressed);

var x = 0;
var y = 0;
for(var i = 0; i < 100; i++)
{
	x++;
	y++;
	plotWidget.addDataToGraph(plotWidgetGraph1Index,  x,  y);
	plotWidget.addDataToGraph(plotWidgetGraph2Index,  x + 10,  y);
	if(y > 10)
	{
```

```
                y = 0;
        }
}
```

Script Canvas2D

This class provides a subset of the HTML Canvas 2D Context object (https://www.w3.org/TR/2dcontext.)

To create a ScriptCanvas2DWidget following must be done:

- add a group box to the script GUI (QtDesigner)
- call ScriptGroupBox::addCanvas2DWidget (see chapter ScriptPlotWidget addPlotWidget(void))

Important: This class is **not** derived from the ScriptWidget class. Therefore the functions from the ScriptWidget class can not be used.
The functions and signals which can be used from script are described in the following chapters.

qreal globalAlpha

This read/write property holds the current alpha value applied to rendering operations.
The value must be in the range from 0.0 (fully transparent) to 1.0
(fully opaque). The default value is 1.0.

QString globalCompositeOperation

This read/write property holds the current the current composition operation.
The default value is source-over. Following values are possible:

- "source-over"
- "destination-over"
- "clear"
- "source"
- "destination"
- "source-in"
- "destination-in"
- "source-out"
- "destination-out"
- "source-atop"
- "destination-atop"
- "xor"
- "plus"
- "multiply"
- "screen"
- "overlay"
- "darken"
- "lighten"
- "color-dodge"
- "color-burn"
- "hard-light"

- "soft-light"
- "difference"
- "exclusion"

Note: See QPainter::CompositionMode for more details.

QVariant strokeStyle

This read/write property holds the current color or style to use for the lines around shapes.
The style can be either a string containing a CSS color, a CanvasGradient or CanvasPattern object. Invalid values are ignored. The default value is '#000000'.

QVariant fillStyle

This read/write property holds the current style used for filling shapes. The style can be either a string containing a CSS color, a CanvasGradient or CanvasPattern object- The default value is '#000000'.

qreal lineWidth

This read/write property holds the current line width. Values that are not finite values greater than zero are ignored. The default value is 1.

QString lineCap

This read/write property holds the current line cap style. The possible line cap styles are:

- butt: the end of each line has a flat edge perpendicular to the direction of the line, this is the default line cap value.
- round: a semi-circle with the diameter equal to the width of the line must then be added on to the end of the line.
- square: a rectangle with the length of the line width and the width of half the line width, placed flat against the edge perpendicular to the direction of the line.

QString lineJoin

This read/write property holds the current line join style. A join exists at any point in a subpath shared by two consecutive lines. When a subpath is closed, then a join also exists at its first point (equivalent to its last point) connecting the first and last lines in the subpath. The possible line join styles are:

- bevel: this is all that is rendered at joins.
- round: a filled arc connecting the two aforementioned corners of the join, abutting (and not overlapping) the aforementioned triangle, with the diameter equal to the line width and the origin at the point of the join, must be rendered at joins.
- miter: a second filled triangle must (if it can given the miter length) be rendered at the join, this is the default line join style.

qreal miterLimit

This read/write property holds the current miter limit ratio. The default miter limit value is 10.0.

qreal shadowOffsetX
This read/write property holds the current shadow offset in the positive horizontal distance. The default value is 0.

qreal shadowOffsetY
This read/write property holds the current shadow offset in the positive vertical distance. The default value is 0.

qreal shadowBlur
This read/write property holds the current level of blur applied to shadows. The default value is 0.

QString shadowColor
This read/write property holds the current shadow color. The default value is '#000000'.

QString width
This read property holds the width of the canvas widget.

QString height
This read property holds the height of the canvas widget.

void save()
Pushes the current state onto the state stack.
Before changing any state attributes, you should save the current state for future reference. The context maintains a stack of drawing states. Each state consists of the current transformation matrix, clipping region, and values of the following attributes:

- strokeStyle
- fillStyle
- globalAlpha
- lineWidth
- lineCap
- lineJoin
- miterLimit
- shadowOffsetX
- shadowOffsetY
- shadowBlur
- shadowColor
- globalCompositeOperation

void restore()
Pops the top state on the stack, restoring the context to that state.

void scale(qreal x, qreal y)
Increases or decreases the size of each unit in the canvas grid by multiplying the scale factors to the current tranform matrix. x is the scale factor in the horizontal direction and y is the scale factor in the vertical direction.

void rotate(qreal angle)
Rotate the canvas around the current origin by angle in radians and clockwise direction.

void translate(qreal x, qreal y)
Translates the origin of the canvas by a horizontal distance of x, and a vertical distance of y, in coordinate space units. Translating the origin enables you to draw patterns of different objects on the canvas without having to measure the coordinates manually for each shape.

void setTransform(qreal m11, qreal m12, qreal m21, qreal m22, qreal dx, qreal dy)
Changes the transformation matrix to the matrix given by the arguments as described below. Modifying the transformation matrix directly enables you to perform scaling, rotating, and translating transformations in a single step. Each point on the canvas is multiplied by the matrix before anything is drawn. For more details see the HTML Canvas 2D Context specification.

void transform(qreal m11, qreal m12, qreal m21, qreal m22, qreal dx, qreal dy)
This method is very similar to setTransform(), but instead of replacing the old transform matrix, this method applies the given tranform matrix to the current matrix by multiplying to it.

CanvasGradient createLinearGradient(qreal x0, qreal y0, qreal x1, qreal y1)
Returns a CanvasGradient object that represents a linear gradient that transitions the color along a line between the start point (x0, y0) and the end point (x1, y1). A gradient is a smooth transition between colors. There are two types of gradients: linear and radial. Gradients must have two or more color stops, representing color shifts positioned from 0 to 1 between to the gradient's starting and end points or circles.

CanvasGradient createRadialGradient(qreal x0, qreal y0, qreal r0, qreal x1, qreal y1, qreal r1)
Returns a CanvasGradient object that represents a radial gradient that paints along the cone given by the start circle with origin (x0, y0) and radius r0, and the end circle with origin (x1, y1) and radius r1.

void clearRect(qreal x, qreal y, qreal w, qreal h)
Clears all pixels on the canvas in the given rectangle to transparent black.

void fillRect(qreal x, qreal y, qreal w, qreal h)
Paint the specified rectangular area using the fillStyle.

void strokeRect(qreal x, qreal y, qreal w, qreal h)
Stroke the specified rectangle's path using the strokeStyle, lineWidth, lineJoin, and (if appropriate) miterLimit attributes.

void beginPath()
Resets the current path to a new path.

void closePath()
Closes the current subpath by drawing a line to the beginning of the subpath, automatically starting a new path. The current point of the new path is the previous subpath's first point.

void moveTo(qreal x, qreal y)
Creates a new subpath with the given point.

void lineTo(qreal x, qreal y)
Draws a line from the current position to the point (x, y).

void quadraticCurveTo(qreal cpx, qreal cpy, qreal x, qreal y)
Adds a quadratic bezier curve between the current point and the endpoint (x, y) with the control point specified by (cpx, cpy).

void bezierCurveTo(qreal cp1x, qreal cp1y, qreal cp2x, qreal cp2y, qreal x, qreal y)
Adds a cubic bezier curve between the current position and the given endPoint using the control points specified by (cp1x, cp1y) and (cp2x, cp2y). After the curve is added, the current position is updated to be at the end point (x, y) of the curve.

void arcTo(qreal x1, qreal y1, qreal x2, qreal y2, qreal radius)
Adds an arc with the given control points and radius to the current subpath, connected to the previous point by a straight line.

void rect(qreal x, qreal y, qreal w, qreal h)
Adds a rectangle at position (x, y), with the given width w and height h, as a closed subpath.

void arc(qreal x, qreal y, qreal radius, qreal startAngle, qreal endAngle, bool anticlockwise)
Adds an arc to the current subpath that lies on the circumference of the circle whose center is at the point (x, y) and whose radius is radius. Both startAngle and endAngle are measured from the x-axis in radians. The anticlockwise parameter is true for each arc in the figure above because they are all drawn in the anticlockwise direction.

void fill()
Fills the subpaths with the current fill style.

void stroke()
Strokes the subpaths with the current stroke style.

void clip()
Creates the clipping region from the current path. Any parts of the shape outside the clipping path are not displayed.

bool isPointInPath(qreal x, qreal y)
Returns true if the given point is in the current path.

void clear()
Clears the canvas widget.

void reset()
Resets the canvas widget.

bool saveToFile(QString fileName, QString imageType="")
Save the canvas widget to an image file. If imageType is empty then the image format will be detected by inspecting the extension of fileName.
Arguments:

- fileName: The file name
- imageType: The image type. Following types are supported.
    - BMP (Windows Bitmap)
    - JPG (Joint Photographic Experts Group)
    - PNG (Portable Network Graphics)
    - PBM (Portable Bitmap)
    - PGM (Portable Graymap)
    - PPM (Portable Pixmap)

Return: True on success.

void print(QString printDialogTitle="")
Opens a print dialog and prints the canvas widget.
Arguments:

- printDialogTitle: The title of the print dialog

Script Canvas2D Example
An example can be found under exampleScripts\WorkerScripts\Canvas2D.

ScriptWidget class
This is the parent class of almost all user interface classes.
The functions and signals which can be used from the worker script are described in the following chapters.

void setEnabled(bool isEnabled)
Enables or disables the widget.

void update(void)
Updates the widget.

void repaint (void)
Repaints the widget.

void show (void)
Shows the widget.

void close (void)
Closes the widget.

void hide (void)
Hides the widget.

void setWindowTitle(QString title)
Sets the window title.
Arguments:

- title: The new title

QString windowPositionAndSize(void)
Returns the window size and position (Pixel). The return string has following format: "top left x, top left y, width, height".

void setWindowPositionAndSize(QString positionAndSize)
Sets the position and the size of a window (Pixel). String format: "top left x, top left y, width, height".
Arguments:

- positionAndSize: The position and size string

Example:
```
Ui_Dialog.setWindowPositionAndSize("100,100,500,500");
```

void setBackgroundColor(QString color)
Sets the background color of a script gui element.
Arguments:

- color: The color, possible values are: black, white, gray, red, green, blue, cyan, magenta and yellow

Example:
```
UI_testSetTextLineEdit.setBackgroundColor("red");
```

void setWindowTextColor(QString color)
Sets the window text color of a script gui element.
Arguments:

- color: The color, possible values are: black, white, gray, red, green, blue, cyan, magenta and yellow

Example:
```
UI_testReceiveCheckBox.setWindowTextColor("red");
```

void setTextColor(QString color)
Sets the text color of a script gui element.
Arguments:

- color: The color, possible values are: black, white, gray, red, green, blue, cyan, magenta and yellow

Example:
```
UI_testReceiveCheckBox.setTextColor("red");
```

void setPaletteColor(QString palette, QString color)

Sets a palette color of a script gui element.
Arguments:

- palette: The palette element, possible values are: Base, Foreground, Background, WindowText, Window, Text and ButtonText.
- color: The color, possible values are: black, white, gray, red, green, blue, cyan, magenta and yellow

Example:

```
UI_testReceiveCheckBox.setPaletteColor("Text", "red");
```

void setPaletteColorRgb(quint8 red, quint8 green, quint8 blue, QString palette)

Sets a palette color of a script gui element.
Arguments:

- red: the red value
- green: the green value
- blue: the blue value
- palette: The palette element, possible values are: Base, Foreground, Background, WindowText, Window, Text and ButtonText.

Example:

```
UI_testReceiveCheckBox.setPaletteColorRgb(255, 255, 255, "Text");
```

void setToolTip(QString text, int duration)

Sets the tool tip of the script gui element.
If the duration is -1 (default) the duration is calculated depending on the length of the tool tip.
Arguments:

- text: The tool tip text
- duration: The tool tip duration

Example:

```
UI_testGetTextLineEdit.setToolTip("tool tip text", -1);
```

void raise(void)

Raises this widget to the top of the parent widget's stack.

void lower(void)

Lowers the widget to the bottom of the parent widget's stack.

void setWindowFlags(quint32 flags)

Sets the window flags.
Window flags are a combination of a type (e.g. Qt::Dialog) and zero or more hints to the window system (e.g. Qt::FramelessWindowHint).
If the widget had type Qt::Widget or Qt::SubWindow and becomes a window (Qt::Window, Qt::Dialog, etc.), it is put at position (0, 0) on the desktop. If the widget is a window and becomes a Qt::Widget or Qt::SubWindow, it is put at position (0, 0) relative to its parent widget.

Note: This function calls setParent() when changing the flags for a window, causing the widget to be hidden. You must call **ScriptWidget::show()** to make the widget visible again.

Arguments:

- flags: The new additional window flags

The flag argument is a Qt enumeration. It is defined as:

```
enum WindowType
{
        Widget = 0x00000000,
        Window = 0x00000001,
        Dialog = 0x00000002 | Window,
        Sheet = 0x00000004 | Window,
        Drawer = Sheet | Dialog,
        Popup = 0x00000008 | Window,
        Tool = Popup | Dialog,
        ToolTip = Popup | Sheet,
        SplashScreen = ToolTip | Dialog,
        Desktop = 0x00000010 | Window,
        SubWindow = 0x00000012,
        ForeignWindow = 0x00000020 | Window,
        CoverWindow = 0x00000040 | Window,

        WindowType_Mask = 0x000000ff,
        MSWindowsFixedSizeDialogHint = 0x00000100,
        MSWindowsOwnDC = 0x00000200,
        BypassWindowManagerHint = 0x00000400,
        X11BypassWindowManagerHint = BypassWindowManagerHint,
        FramelessWindowHint = 0x00000800,
        WindowTitleHint = 0x00001000,
        WindowSystemMenuHint = 0x00002000,
        WindowMinimizeButtonHint = 0x00004000,
        WindowMaximizeButtonHint = 0x00008000,
        WindowMinMaxButtonsHint = WindowMinimizeButtonHint |
                                  WindowMaximizeButtonHint,
        WindowContextHelpButtonHint = 0x00010000,
        WindowShadeButtonHint = 0x00020000,
        WindowStaysOnTopHint = 0x00040000,
        WindowTransparentForInput = 0x00080000,
        WindowOverridesSystemGestures = 0x00100000,
        WindowDoesNotAcceptFocus = 0x00200000,

        CustomizeWindowHint = 0x02000000,
        WindowStaysOnBottomHint = 0x04000000,
        WindowCloseButtonHint = 0x08000000,
        MacWindowToolBarButtonHint = 0x10000000,
        BypassGraphicsProxyWidget = 0x20000000,
        WindowOkButtonHint = 0x00080000,
        WindowCancelButtonHint = 0x00100000,
        NoDropShadowWindowHint = 0x40000000,
        WindowFullscreenButtonHint = 0x80000000
    };
```

Example:

```
UI_ReceiveFileDialog.setWindowFlags(0x00040000);
//show must be called after setWindowFlags
UI_ReceiveFileDialog.show();
```

quint32 windowFlags(void)

Returns the window flags.

void clearWindowFlags(quint32 flags)

Clears the given window flags.

Note: ScriptWidget::show must be called after a clearWindowFlags call.

Arguments:

- flags: The window flags which shall be cleared (for more details about window flags see void setWindowFlags(quint32 flags)

Example:

```
UI_ReceiveFileDialog.clearWindowFlags(0x00040000);
//show must be called after setWindowFlags
UI_ReceiveFileDialog.show();
```

void setFocus(void)

Gives the keyboard input focus to this widget.

void width(void)

Returns the width of the widget excluding any window frame.

void height(void)

Returns the height of the widget excluding any window frame.

QWidget getWidgetPointer(void)*

Returns the widget pointer.

Example:

```
var input = scriptThread.showTextInputDialog("Title", "label", "initial text",
UI_Dialog.getWidgetPointer())
```

void setAdditionalData(int key, QString data)

Sets/stores an additional data entry (internally stored in a QMap<int, QString> map).

Arguments:

- key: The data key
- data: The data string

QString getAdditionalData(int key)

Returns an additional data entry (internally stored in a QMap<int, QString> map).

Arguments:

- key: The data key

QString getClassName(void)

Returns the class name of this object.

bool blockSignals(bool block)

If block is true, signals emitted by this object are blocked (i.e., emitting a signal will not invoke anything connected to it). If block is false, no such blocking will occur.

The return value is the previous value of the blocking state.

Custom script widget

ScriptCommunicator can be extended with a custom script widget. For this following must be done:

- create a Qt widget (see the Qt documentation)
- create a QtDesigner plug-in (see the Qt documentation)
- export the functions GetScriptCommunicatorWidgetName and CreateScriptCommunicatorWidget
- create a wrapper class through which worker scripts can access the custom widget (the custom widget resides (like a GUI elements) in the main thread and worker threads resides in their own thread)
- put the QtDesigner plug-in library into the plugins\designer folder

Note: All functions and classes must reside in the QtDesigner plug-in library.

extern "C" Q_DECL_EXPORT const char GetScriptCommunicatorWidgetName(void)*

Returns the class name of the custom widget.

*extern "C" Q_DECL_EXPORT QObject *CreateScriptCommunicatorWidget(QObject *scriptThread, QWidget *customWidget, bool scriptRunsInDebugger)*

Creates the wrapper class through which a worker-script can access the custom widget.
Arguments:

- scriptThread: Pointer to the script thread.
- customWidget: The custom widget.
- scriptRunsInDebugger: True if the script thread runs in a script debugger.

Important:
A function from a custom widget class must not be called directly from a worker-script (a worker script runs in his own thread and the custom widget runs like all GUI elements in the main thread). Instead a slot must be called (by a signal). This signal must be connected with Qt::QueuedConnection or Qt::BlockingQueuedConnection.
If the worker script runs in a debugger then Qt::DirectConnection instead of Qt::BlockingQueuedConnection must be used (the debugger and therefore the worker script runs in the main thread). Qt::BlockingQueuedConnection would cause a dead-lock.

Examples can be found under exampleScripts\WorkerScripts\CustomWidget.

Dynamic link libraries

Worker script can extend their functionality by loading a dynamic link library. This library must export following function:

```
extern "C" Q_DECL_EXPORT void init(QScriptEngine* engine);
```

In this function all objects and functions which are accessible by script must be registered.

C++ Example:

```
void init(QScriptEngine* engine)
{
    qRegisterMetaType<QScriptEngine*>("TestDll*");
    engine->globalObject().setProperty("TestDll", engine->newQObject(&testDll));

    QScriptValue sendDataArrayFunction = engine->evaluate("sendDataArray");
    testDll.setSendDataArrayFunction(sendDataArrayFunction);
```

```
    testDll.setScriptEngine(engine);
}
```

To load a library the script must call the function bool loadLibrary(QString path, bool isRelativePath=true)
Example:

```
scriptThread.loadLibrary("TestDll.dll", true);
```

Under exampleScripts/WorkerScripts/LoadLibrary a script and a Qt library Project can be found which demonstrate the usage.

Sequence script

Sequence scripts can be added to a send sequence (send window). If a sequence is send, the script function sendData is called. In this function the sequence data can be modified (create a frame, calculate CRC …). The modified data must be returned.

Note: After sending a sequence the corresponding sequence script is unloaded. Therefore no data can be stored in a sequence script variable (sequence scripts can store data globally in ScriptCommunicator (e.g. counter) with special functions (are described below)).
Example:

```
function sendData(data)
{
	var counter = 0;
	//Read the stored counter value
	var resultArray = seq.getGlobalUnsignedNumber("Counter");
	if(resultArray[0] == 1)
	{
		counter = resultArray[1];
	}

	//Append the counter value.
	data.push((counter >> 24) & 0xff);
	data.push((counter >> 16) & 0xff);
	data.push((counter >> 8) & 0xff);
	data.push(counter & 0xff);

	//Append a CRC8.
	var crc8 = seq.calculateCrc8(data);
	data.push(crc8 & 0xff);

	counter++;
	//Store the new counter value.
	seq.setGlobalUnsignedNumber("Counter", counter);

	return data;
}
```

Note:
For complex scripts 'worker scripts' in the script window must be used (see chapter Worker scripts).

Sequence scripts are running in an own thread, therefore ScriptCommunicator can not be blocked by a sequence script directly. The GUI of a sequence script runs in the main thread (a call to a script GUI element normally calls a function in the main thread), therefore to many calls to script GUI elements

can block ScriptCommunicator.

Sequence scripts are QtScript scripts (see chapter Script interface). The sequence scripts interface extents the standard QtScript functionality. These extended functionality is described in the following chapters.

QVector<unsigned char> data sendData(QVector<unsigned char> data)

This function is called if a sequence shall be sent (every sequence script must contain this function).
Arguments:

- data: The sequence data array

Return: The modified data array.
Note: The sequence will not be sent if an empty array (Array()) is returned.

QString byteArrayToString (QVector<unsigned char> data)

Converts a byte array which contains ASCII characters into an ASCII string.
Example:

```
var array = Array(48, 49, 50);
var string = seq.byteArrayToString(array);
```

QString byteArrayToHexString (QVector<unsigned char> data)

Converts a byte array into a hex string.
Example:

```
var array = Array(2, 3, 4, 33);
var string = seq.byteArrayToHexString(array);
```

QVector<unsigned char> stringToArray(QString str)

Converts an ASCII string into a byte array.
Example:

```
var array = seq.stringToArray("Test");
```

QVector<unsigned char> addStringToArray(QVector<unsigned char> array, QString str)

Adds an ASCII string to a byte array.
Example:

```
var array = Array(0,1,2,3,4);
array = seq.addStringToArray(array, "Test");
```

QString showTextInputDialog(QString title, QString label, QString displayedText="")

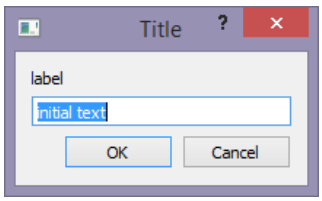
Convenience function to get a string from the user. Shows a QInputDialog::getText dialog (line edit).
Arguments:

- title: The title of the dialog
- label: The label over the input area
- displayedText: The initial displayed text in the input area

Return: The text in the input section after closing the dialog (empty if the ok button was not pressed).
Example:

```
var input = seq.showTextInputDialog("Title", "label", "initial text");
if(input != "")
{//OK button pressed.
	//Process the string in 'input'.
```

```
}
else
{//OK button not pressed or empty input.
}
```



QString showMultiLineTextInputDialog(QString title, QString label, QString displayedText="")

Convenience function to get a multiline string from the user. Shows a QInputDialog::getMultiLineText dialog (plain text edit).

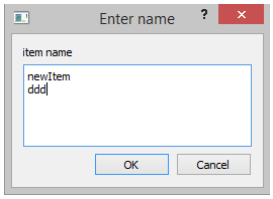
Arguments:

- title: The title of the dialog
- label: The label over the input area
- displayedText: The initial displayed text in the input area
- parent: The parent of this dialog (see chapter QWidget* getWidgetPointer(void) for more details)

Return: The text in the input section after closing the dialog (empty if the ok button was not pressed).

Example:

```
var input = seq.showMultiLineTextInputDialog("Enter name", "item name", "newItem");
if(input != "")
{//OK button pressed.
	//Process the string in 'input'.
}
else
{//OK button not pressed or empty input.
}
```



QString showGetItemDialog(QString title, QString label, QStringList displayedItems, int currentItemIndex=0, bool editable=false, QWidget parent=0)*

Convenience function to let the user select an item from a string list. Shows a QInputDialog::getItem dialog (combobox).

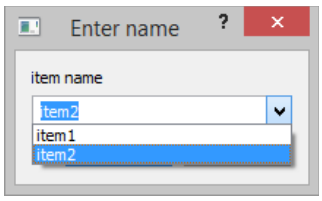
Arguments:

- title: The title of the dialog
- label: The label over the input area
- displayedItems: The displayed items
- currentItemIndex: The current combobox index
- editable: True if the combobox shall be editable
- parent: The parent of this dialog (see chapter QWidget* getWidgetPointer(void) for more details)

Return: The text of the selected item after closing the dialog (empty if the ok button was not pressed).

Example:

```
var input = seq.showGetItemDialog("Enter name", "item name", Array("item1",
"item2"), 1, true);
if(input != "")
{//OK button pressed.
	//Process the string in 'input'.
}
else
{//OK button not pressed or empty input.
}
```



QList<int> showGetIntDialog(QString title, QString label, int intialValue, int min, int max, int step, QWidget parent=0)*

Convenience function to get an integer input from the user. Shows a QInputDialog::getInt dialog (spinbox).

Arguments:

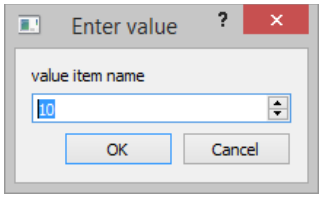
- title: The title of the dialog
- label: The label over the input area
- intialValue: The initial value.
- min: The minimum value
- max: The maximum value.
- step: The amount by which the values change as the user presses the arrow buttons to increment or decrement the value
- parent: The parent of this dialog (see chapter QWidget* getWidgetPointer(void) for more details)

Return:

- array item 0: 1 if the ok button has been pressed, 0 otherwise
- array item 1: The value of the spinbox after closing the dialog

Example:

```
var resultArray = seq.showGetIntDialog("Enter value", "value item name", 10, 0, 20,
2);
if(resultArray[0] == 1)
{//OK button pressed.
	//Process the string in 'input'.
}
else
{//OK button not pressed or empty input.
}
```



QList<double> showGetDoubleDialog(QString title, QString label, double intialValue, double min, double max, int decimals, QWidget parent)*

Convenience function to get a floating point number from the user. Shows a QInputDialog::getDouble dialog (spinbox).

Arguments:

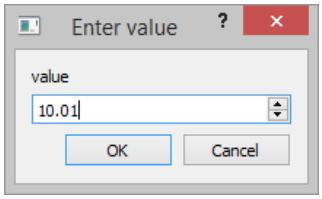
- title: The title of the dialog
- label: The label over the input area
- intialValue: The initial value.
- min: The minimum value
- max: The maximum value.
- decimals: The maximum number of decimal places the number may have
- parent: The parent of this dialog (see chapter QWidget* getWidgetPointer(void) for more details)

Return:

- array item 0: 1.0 if the ok button has been pressed, 0 otherwise
- array item 1: The value of the spinbox after closing the dialog

Example:

```
var resultArray = seq.showGetDoubleDialog("Enter value", "value", 10, 0, 20, 2);
if(resultArray[0] >= 1.0)
{//OK button pressed.
	//Process the string in 'input'.
}
else
{//OK button not pressed or empty input.
}
```



void messageBox(QString icon, QString title, QString text, QWidget parent=0)*

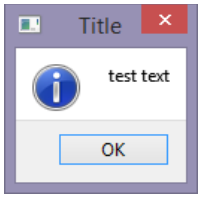
This function shows a message box.

Arguments:

- icon: The icon of the message box. Possible values are: "Information"," Warning", "Critical" and "Question"
- title: The title of the message box
- text: The text of the message box
- parent: The parent of this dialog (see chapter QWidget* getWidgetPointer(void) for more details)

Example:

```
seq.messageBox("Information", "Title", "test text");
```



bool showYesNoDialog(QString icon, QString title, QString text, QWidget parent=0)*

This function shows a yes/no dialog.

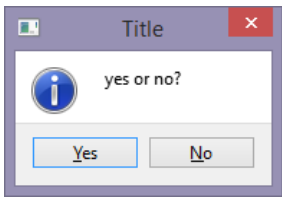
Arguments:

- icon: The icon of the dialog. Possible values are: "Information"," Warning", "Critical" and "Question"
- title: The title of the dialog
- text: The text of the dialog
- parent: The parent of this dialog (see chapter QWidget* getWidgetPointer(void) for more details)

Return: True if the user has pressed the yes button

Example:

```
if(seq.showYesNoDialog("Information", "Title", "yes or no?"))
{//Yes clicked.
	//Do something.
}
```



QList<int> showColorDialog(quint8 initialRed=255, quint8 initialGreen=255, quint8 initialBlue=255, quint8 initialAlpha=255, bool alphaIsEnabled=false)

Convenience function to get color settings from the user.

Arguments:

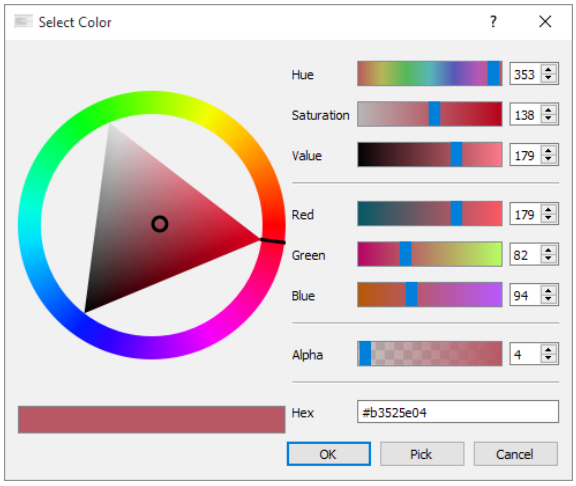
- initialRed: The initial value for red
- initialGreen: The initial value for green
- initialBlue: The initial value for blue
- initialAlpha: The initial value for alpha
- alphaIsEnabled: True if the alpha value should be visible/editable

Return: integer list which contains:

- element 0: 1 = OK button press, 0 OK button not pressed
- element 1: red (0-255)
- element 2: green (0-255)
- element 3: blue (0-255)
- element 4: alpha (0-255)

Example:

```
var resultArray = seq.showColorDialog(1,2,3,4,true);
var data = Array();
if(resultArray[0])
{//OK clicked.
	data.push(resultArray[1]); //Red
	data.push(resultArray[2]); //Green
	data.push(resultArray[3]); //Blue
	data.push(resultArray[4]); //Alpha
}
```



quint8 calculateCrc8(QVector<unsigned char> data)

Calculates a CRC8.

Arguments:

- data: The data for calculating the CRC

Return: The CRC

Example:

```
var dataArray = Array(0, 1, 2, 3, 4, 5, 6, 7)
var crc =  seq.calculateCrc8(dataArray);
```

The used code for the CRC calculation is shown on page 42.

quint16 calculateCrc16(QVector<unsigned char> data)

Calculates a CRC16.

Arguments:

- data: The data for calculating the CRC

Return: The CRC

Example:

```
var dataArray = Array(0, 1, 2, 3, 4, 5, 6, 7)
var crc =  seq.calculateCrc16(dataArray);
```

The used code for the CRC calculation is shown on page 43.

quint32 calculateCrc32(QVector<unsigned char> data)

Calculates a CRC32.

Arguments:

- data: The data for calculating the CRC

Return: The CRC

Example:

```
var dataArray = Array(0, 1, 2, 3, 4, 5, 6, 7)
var crc =  seq.calculateCrc32(dataArray);
```

The used code for the CRC calculation is shown on page 44.

quint64 calculateCrc64(QVector<unsigned char> data)

Calculates a CRC64.

Arguments:

- data: The data for calculating the CRC

Return: The CRC

Example:

```
var dataArray = Array(0, 1, 2, 3, 4, 5, 6, 7)
var crc =  seq.calculateCrc64(dataArray);
```

The used code for the CRC calculation is shown on page 45.

QString getCurrentVersion

Returns the current version of ScriptCommunicator (string) .

Version format: major.minor (eg. 3.09)

void setBlockTime(quint32 blockTime)

Sets the script block time.

Note: After this execution time (sendData and the script main function (all outside a function))

the script is regarded as blocked and will be stopped. The default is 10000.

Inter-SequenceScript communication

The following functions can be used to store data from a sequence script globally. If this sequence script is started again the data can be read (e.g. for a counter). Global stored sequence script data\variables can be accessed from every sequence script (these variable are stored in ScriptCommunicator global sequence script maps).

Under exampleScripts\SequenceSendScript an example of global sequence variables can be found.

void setGlobalString(QString name, QString string)

Sets a string in the global string map.
Arguments:

- name: Name of the string variable
- string: The string

QString getGlobalString(QString name, bool removeValue=false)

Returns a string from the global string map.
Arguments:

- name: Name of the variable
- removeValue: True if the variable shall be removed from the string map

Return: The read string. Returns an empty string if name is not in the string map.

void setGlobalDataArray(QString name, QVector<unsigned char> data)

Sets a data array in the global data array map.
Arguments:

- name: Name of the variable
- data: The data array

QVector<unsigned char> getGlobalDataArray(QString name, bool removeValue=false)

Returns a data array from the global data array map.
Arguments:

- name: Name of the variable
- removeValue: True if the variable shall be removed from the data array map

Return: The read data array. Returns an empty data array if name is not in the data array map.

void setGlobalUnsignedNumber(QString name, quint32 number)

Sets an unsigned number in the global unsigned number map.
Arguments:

- name: Name of the variable
- number: The number

QList<quint32> getGlobalUnsignedNumber(QString name,bool removeValue=false)

Returns an unsigned number from the global unsigned number map.
Arguments:

- name: Name of the variable
- removeValue: True if the variable shall be removed from the unsigned number map

Return: The first element in the result list is the result status (1=name found, 0=name not found). The second element is the read value.
Example:

```
var counter = 0;
//Read the stored counter value
var resultArray = seq.getGlobalUnsignedNumber("Counter");
if(resultArray[0] == 1)
{
	counter = resultArray[1];
}
```

void setGlobalSignedNumber(QString name, quint32 number)

Sets a signed number in the global signed number map.
Arguments:

- name: Name of the variable
- number: The number

QList<quint32> getGlobalSignedNumber(QString name,bool removeValue=false)

Returns a signed number from the global signed number map.
Arguments:

- name: Name of the variable
- removeValue: True if the variable shall be removed from the signed number map

Return: The first element in the result list is the result status (1=name found, 0=name not found). The second element is the read value.
Example:

```
var counter = 0;
//Read the stored counter value
var resultArray = seq.getGlobalSignedNumber("Counter");
if(resultArray[0] == 1)
{
	counter = resultArray[1];
}
```

Custom console/log scripts

Custom console/log scripts can be added in the settings dialog (console and log tab). With this scripts a custom console and a custom log file can be created which have a custom format.

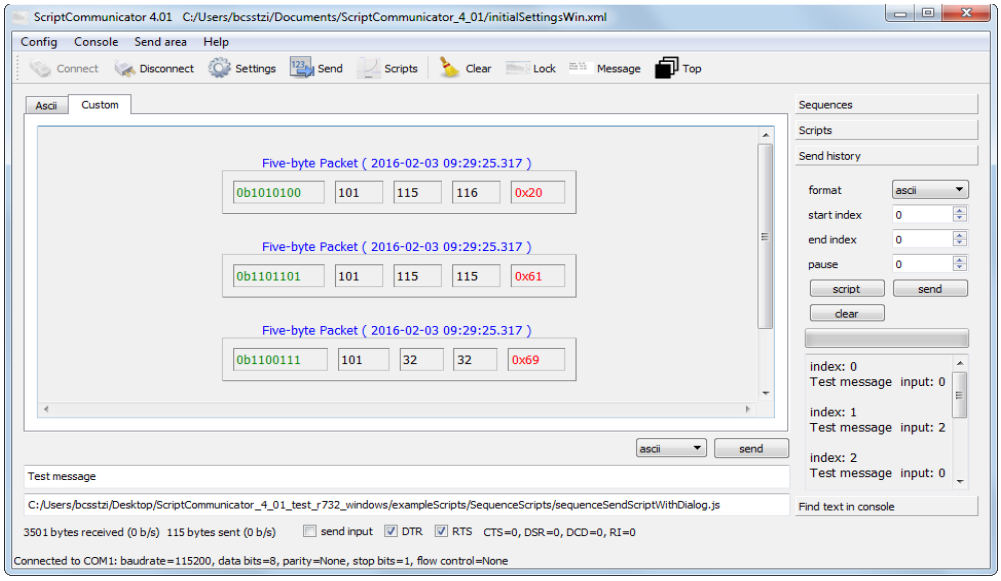
If data has been received/sent or a user message has been entered (message dialog or worker thread) the script function createString is called. This function must return a string which will be added to the custom console or the custom log.
Example:

```
var storedData = Array()
function createString(data, timeStamp, type, isLog)
{
	var resultString = "";
	appendByteArrayAtByteArray(storedData, data, data.length);

	//After 5 bytes have been reived the packet table is created.
	if(storedData.length >=5)
	{
		resultString = '<TABLE ALIGN=CENTER WIDTH="50%" BORDER=1
	CELLSPACING=10 CELLPADDING=3><CAPTION><p style="color:blue">Five-byte Packet
```

```
        ('+timeStamp+')</p></CAPTION><TR><TD> <p
        style="color:green">0b'+storedData[0].toString(2)+'</p></TD> <TD>
        '+storedData[1]+'</TD> <TD> '+storedData[2]+'</TD><TD> '+storedData[3]+'</TD>
        <TD> <p style="color:red"> 0x'+storedData[4].toString(16).toUpperCase()
        +'</p></TD> </TR></TABLE> <BR>';
               storedData.splice(0, 5);
        }
        return resultString;
}
function appendByteArrayAtByteArray(dest, source, maxBytes)
{
        for(var i = 0; i < source.length; i++)
        {
               if(maxBytes <= i )
               {//To many bytes in source.
                      break;
               }
               dest.push(source[i]);
        }
}
```

This script generates following output:



Under exampleScripts/CustomLogConsoleScripts several example custom console/log scripts can be found.

A custom console/log script runs in his own thread, therefore ScriptCommunicator can not be blocked by a custom console/log script.

Custom console/log scripts are QtScript scripts (see chapter Script interface). The Custom console/log script interface extents the standard QtScript functionalities. These extended functionality is described in the following chapters.

Note:

Custom console/log scripts are loaded at the time when they are selected in the settings dialog or when ScriptCommunicator starts. They are unload when they are deselected in the settings dialog or

when ScriptCommunicator exits. Therefore custom console/log scripts can store data (variables) inside the script (in contrast to a sequence script).

string createString(data, timeStamp, type, isLog)

This function is called if:

- data has been sent
- data has been received
- a user message has been entered (from the message dialog or a worker script (scriptThread.addMessageToLogAndConsoles))

Here the string is created which shall be added to the custom console or to the custom log (argument isLog)
Note: The custom console (**QTextEdit is used**) interprets the returned text as HTML (if a new line shall be created, then a <br> must be returned (and not \n)).
Therefore every created console string can have its own format (text color, text size, font family, ...).
If no format information is given then the format settings from the settings dialog are used (text color=receive color).
The created log strings are directly (without interpreting the content) written into the custom log file.
If the data is from a CAN interface then the bytes have the following meaning:

- Byte 0: message type (0=standard, 1=standard remote-transfer-request, 2=extended, 3=extended remote-transfer-request)
- Byte 1-4: can id
- Byte 5-12: the data.

Arguments:

- data: the data
- timeStamp: the time stamp string (the format is set in the settings dialog).
- Type: the data type:
  - 0=the data has been received from a normal interface (all but CAN)
  - 1=the data has been sent with a normal interface (all but CAN)
  - 2=the data has been received from the CAN interface
  - 3=the data has been sent with CAN the can interface
  - 4=the data is a user message (from message dialog or normal script
- isLog: True if this call is for the custom log (false=custom console)

Return:
The created string which will be added to the custom log or the custom console (argument isLog).

QString byteArrayToString (QVector<unsigned char> data)

Converts a byte array which contains ASCII characters into an ASCII string.
Example:

```
var array = Array(48, 49, 50);
var string = cust.byteArrayToString(array);
```

QString byteArrayToHexString (QVector<unsigned char> data)

Converts a byte array into a hex string.
Example:

```
var array = Array(2, 3, 4, 33);
var string = cust.byteArrayToHexString(array);
```

QVector<unsigned char> stringToArray(QString str)

Converts an ASCII string into a byte array.

Example:

```
var array = cust.stringToArray("Test");
```

QVector<unsigned char> addStringToArray(QVector<unsigned char> array, QString str)

Adds an ASCII string to a byte array.

Example:

```
var array = Array(0,1,2,3,4);
array = cust.addStringToArray(array, "Test");
```

QString getScriptFolder(void)

Returns the folder in which the main script resides.

Example:

```
var result = cust.getScriptFolder();
```

QString createAbsolutePath(QString path)

Converts a relative path (relative to the current script) into an absolute path.

Arguments:

- path: the relative path

Return: The absolute path.

Example:

```
var result = cust.createAbsolutePath("file1.txt");
```

bool loadScript(QString scriptPath, bool isRelativePath=true)

Loads/includes one script (QtScript has no built in include mechanism).

Arguments:

- scriptPath: The script path
- isRelativePath: True if the script path is a relative path (relative to the custom console/log script)

Return: True on success.

Example:

```
var result = cust.createAbsolutePath("helper.js");
```

QString getCurrentVersio(void)

Returns the current version of ScriptCommunicator (string) .

Version format: major.minor (eg. 3.09)

Example:

```
var result = cust.getCurrentVersion();
```

void setBlockTime(quint32 blockTime)

Sets the script block time (ms).

Note: After this execution time (createString and the script main function (all outside a function)) the script is regarded as blocked and will be terminated. The default is 10000.

Example:

```
cust.setBlockTime(20000);
```

SQL support

Custom console/log scripts can access SQL databases. The functionality is described in chapter SQL support.

XML support

Custom console/log scripts can access XML files with the ScriptXmlReader and the ScriptXmlWriter classes. These classes are described in chapter XML support.
To create an object of this classes following functions must be used:

ScriptXmlReader createXmlReader(void)*

Creates an XML reader.
Example:

```
var reader = cust.createXmlReader();
```

ScriptXmlWriter createXmlWriter(void)*

Creates an XML writer.
Example:

```
var writer = cust.createXmlWriter();
```

Filesystem

The following function can be used to access the file system.

bool checkFileExists(QString path, bool isRelativePath=true)

Checks if a file exists.
Arguments:

- path: The file path.
- isRelativePath: True if the file path is relative to the current script (which executes this function)

Return: True if the file exists and false if not
Example:

```
var result = cust.checkFileExists("Testfile.txt");
```

qint64 getFileSize(QString path, bool isRelativePath=true)

Returns the size of a file.
Arguments:

- path: The file path
- isRelativePath: True if the file path is relative to the current script (which executes this function)

Return: The file size if the file exists, -1 if the file doesn't exists
Example:

```
var size = cust.getFileSize("Testfile.txt");
```

QString readFile (QString path, bool isRelativePath=true, quint64 startPosition=0, qint64 numberOfBytes=-1)

Reads a text file and returns the content.
Arguments:

- path: The file path

- isRelativePath: True if the file path is relative to the current script (which executes this function)
- startPosition: Start position (the file is read from this position)
- numberOfBytes: The number of bytes which shall be read. If numberOfBytes is < 0 then all bytes from startPosition are read

Return: The file as string

Example:

```
//Read the complete file.
var string = cust.readFile("Testfile.txt");
//Read 20000 bytes from byte 100.
var string2 = cust.readFile("Testfile2.txt", true, 100, 20000);
```

QVector<unsigned char> readBinaryFile (QString path, bool isRelativePath=true, quint64 startPosition=0, qint64 numberOfBytes=-1)

Reads a binary file and returns the content.

Arguments:

- path: The file path
- isRelativePath: True if the file path is relative to the current script (which executes this function)
- startPosition: Start position (the file is read from this position)
- numberOfBytes: The number of bytes which shall be read. If numberOfBytes is < 0 then all bytes from startPosition are read

Return: The file as byte array

Example:

```
//Read the complete file.
var array = cust.readFile("Testfile.bin");
//Read 20000 bytes from byte 100.
var array2 = cust.readFile("Testfile2.bin", true, 100, 20000);
```

bool writeFile(QString path, bool isRelativePath, QString content, bool replaceFile, qint64 startPosition=-1)

Writes a text file.

Arguments:

- path: The file path
- isRelativePath: True if the file path is relative to the current script (which executes this function)
- content: The content to write
- replaceFile: If replaceFile is true, the existing file will be overwritten, else the content is appended
- startPosition: If replaceFile is false then this is the start position at which the data will be written. For appending the data at the end of the file startPosition must be < 0.

Return: True on success

Example:

```
//Replace the file.
var result = cust.writeFile("Testfile.txt", true, "new content", true);
//Append text.
result = cust.writeFile("Testfile.txt", true, "new content", false);
//Write from position 3.
result = cust.writeFile("Testfile.txt", true, "new content", false, 3);
```

bool writeBinaryFile(QString path, bool isRelativePath, QVector<unsigned char> content, bool replaceFile, qint64 startPosition=-1))

Writes a binary file file.

Arguments:

- path: The file path
- isRelativePath: True if the file path is relative to the current script (which executes this function)
- content: The content to write
- replaceFile: If replaceFile is true, the existing file is overwritten, else the content is appended
- startPosition: If replaceFile is false then this is the start position at which the data will be written. For appending the data at the end of the file startPosition must be < 0.

Return: True on success

Example:

```
var array = Array(1,2,3,4,5,6);
//Replace the file.
var result = cust.writeBinaryFile("Testfile.bin", true, array, true);
//Append data.
Result = cust.writeBinaryFile("Testfile.bin", true, array, false);
//Write from position 3.
result = cust.writeBinaryFile("Testfile.bin", true, array, false, 3);
```

bool deleteFile(QString path, bool isRelativePath=true)

Deletes a file.

Arguments:

- path: The file path.
- isRelativePath: True if the file path is relative to the current script (which executes this function)

Return: True on success:

```
var result = cust.deleteFile("Testfile.txt");
```

bool renameFile(QString path, QString newName)

Renames a file.

Arguments:

- path: The file path.
- newName: The new name

Return: True on success:

```
var result = cust.renameFile("C:/Dir1/TestFile.txt", "C:/Dir1/newName.txt");
```

QStringList readDirectory(QString directory, bool isRelativePath=true, bool recursive=true, bool returnFiles=true, bool returnDirectories=true)

Reads the content of a directory and his sub directories.

Arguments:

- directory: The directory path
- isRelativePath: True if the directory path is relative to the current script (which executes this function)
- recursive: If true the result includes the contents of all sub directories (and their sub directories)

- returnFiles: If true the result contains all found files
- returnDirectories: If true the result contains all found directories

Return: The found entries
Example:
```
var array = cust.readDirectory("TestDir");
```

bool checkDirectoryExists(QString path, bool isRelativePath=true)
Checks if a directory exists.
Arguments:
- path: The directory path.
- isRelativePath: True if the directory path is relative to the current script (which executes this function)

Return: True if the directory exists and false if not
Example:
```
var result = cust.checkDirectoryExists("Testdirectory");
```

bool deleteDirectory(QString directory, bool isRelativePath=true)
Deletes a directory (must be empty).
Arguments:
- directory: The directory path.
- isRelativePath: True if the directory path is relative to the current script (which executes this function)

Return: True on success
Example:
```
var result = cust.deletedirectory ("Testdir");
```

bool deleteDirectoryRecursively(QString directory, bool isRelativePath=true)
Removes the directory, including all its contents.
If a file or directory cannot be removed, deleteDirectoryRecursively() keeps going and attempts to delete as many files and sub-directories as possible, then returns false.
If the directory was already removed, the method returns true (expected result already reached).
Arguments:
- directory: The directory path.
- isRelativePath: True if the directory path is relative to the current script (which executes this function)

Return: True on success
Example:
```
var result = cust.deleteDirectoryRecursively ("Testdir");
```

bool createDirectory(QString path, bool isRelativePath=true)
Creates a directory.
Arguments:
- path: The directory path.
- isRelativePath: True if the directory path is relative to the current script (which executes this function)

Return: True on success

Example:

```
var result = cust.createDirectory("Testdirectory");
```

bool renameDirectory(QString path, QString newName,)

Renames a directory.

Arguments:

- path: The directory path.
- newName: The new name (always relative path)

Return: True on success

Example:

```
var result = cust.renameDirectory("C:/Dir1/Testdirectory", "C:/Dir1/newName");
```

SQL support

Worker scripts and custom console/log scripts can access SQL databases. The Qt SQL classes are nearly 1:1 available in ScriptCommunicator. The documentation of the Qt SQL classes can be found here: http://doc.qt.io/qt-5.5/qsqldatabase.html. The differences are described below.

Static QSqlDatabase function

All static QSqlDatabase function are available via the scriptSql Object. Example:

```
var db = scriptSql.addDatabase("QSQLITE");
```

Creating objects

For creating SQL related Objects the scriptSql Object has following functions:

- QSqlQuery createQuery(QSqlDatabase dataBase, QString query = "")
- QSqlField createField()
- QSqlRecord createRecord()

Examples of creating SQL related Objects can be found under:

- exampleScripts\WorkerScripts\TestSql
- exampleScripts\CustomLogConsoleScripts\CustomLogConsole_Sql

Not supported functions

Following functions are not supported:

- QSqlDriver* QSqlDatabase::driver()
- void QsqlDatabase::registerSqlDriver(const QString &name, QSqlDriverCreatorBase *creator)
- QSqlDatabase QSqlDatabase::addDatabase(QSqlDriver* driver, const QString& connectionName = QLatin1String(defaultConnection))
- QsqlDriver* QSqlQuery::driver()
- QsqlResult* QsqlQuery::result()

Mapping of SQL related enumerations

The enumeration are wrapped in quint32 values. These values can be found at the Qt SQL class descriptions. Example:

```
var db = scriptSql.addDatabase("QSQLITE");
//Set the numerical precision policy to QSql::LowPrecisionInt32.
db.setNumericalPrecisionPolicy(0x1);
```

SQL example scripts

Examples scripts which demonstrates the usage of the SQL script functionalities can be found under:

- exampleScripts\WorkerScripts\TestSql
- exampleScripts\CustomLogConsoleScripts\CustomLogConsole_Sql

XML support

Worker scripts and custom console/log scripts can access XML files with the ScriptXmlReader and the ScriptXmlWriter classes. This 2 classes contain a subset of the Qt XML functionality.

Note: Example scripts which demonstrate the usage of the XML script classes class can be found under exampleScripts/WorkerScripts/TestXml and exampleScripts\CustomLogConsoleScripts\CustomLogConsole_Xml.

ScriptXmlReader

Class for reading a xml file.

qint32 readFile(QString fileName, bool isRelativePath=true)

Reads and parses a xml file. The parsed xml file is stored internally.
Arguments:

- fileName: The name of the xml file.
- isRelativePath: True if the file path is relative to the main script.

Return:

0: success
1: file could not be opened
2: parse error

QList<ScriptXmlElement*> elementsByTagName(QString name)

Returns a list containing all xml elements with the name 'name'.
Note: The xml root element is not included.
Arguments:

- name: The name of the xml elements.

Return: A list containing all xml elements.

ScriptXmlElement *getRootElement(void)

Returns the root XML element.

ScriptXmlElement

This class represents a XML element.

QString elementName(void)
Returns the name of this element.

QList<ScriptXmlElement> childElements(void)*
Returns all child elements.

QStringList childTextElements(void)
Returns all child text elements (includes the CDATA elements).

QStringList childCDataElements(void)
Returns all child CDATA elements.

QStringList childCommentElements(void)
Returns all child comment elements.

QString attributeValue(QString attrName)
Returns an attribute value. The attribute is identified by attrName.

QList<ScriptDomAttribute> attributes(void)*
Returns all attributes of this element.

ScriptXmlAttribute
This class represents a xml attributte.

QString value(void)
Returns the value of the attribute.

QString name(void)
Returns the name of the attribute.

ScriptXmlWriter
This class provides functions for creating/writing XML files.
Note: All function are working on an internal XML buffer.
The function writteBufferToFile must be used to write the content of the internal XML buffer to a file.

bool writeBufferToFile(QString fileName, bool isRelativePath=true)
Writes the internal XML Buffer to a file.
Arguments:

- fileName: The name of the xml file.
- isRelativePath: True if the file path is relative to the main script.

Return: True on success.

QString getInternalBuffer(void)
Returns the content of the internal buffer.

QString clearInternalBuffer(void)
Clears the internal buffer.

void setCodec(QString codecName)
Sets the codec for the XML writer to codec. The codec is used for encoding any data that is written. By default, ScriptXmlWriter uses UTF-8. The encoding information is stored in the initial XML tag which gets written when you call writeStartDocument(). Call this function before calling writeStartDocument().
Note: Common values for codecName are "ISO 8859-1", "UTF-8", and "UTF-16". If the encoding isn't recognized, nothing happens.
Arguments:

- codecName: The name of the codec.

void setAutoFormatting(bool autoFormatting)
Sets the autoFormatting property. This property controls whether or not the stream writer automatically formats the generated XML data. If enabled, the writer automatically adds line-breaks and indentation to empty sections between elements (ignorable whitespace). The main purpose of auto-formatting is to split the data into several lines, and to increase readability for a human reader. The indentation depth can be controlled through the autoFormattingIndent property.
Arguments:

- autoFormatting: The new value.

bool autoFormatting(void)
Returns the value of the autoFormating property.

void setAutoFormattingIndent(int spacesOrTabs)
Set the autoFormatingIndent property. This property holds the number of spaces or tabs used for indentation when auto-formatting is enabled. Positive numbers indicate spaces, negative numbers tabs.
Arguments:

- spacesOrTabs: The new value.

int autoFormattingIndent(void)
Returns the autoFormatingIndent property.

void writeStartDocument(QString version="1.0")
Writes a document start with the attribute version.
Arguments:

- version: The XML version.

void writeStartDocument(bool standalone, QString version="1.0")
Writes a document start with the attributes version and standalone.
Arguments:

- standalone: True if the standalone attributes shall be written.
- version: The XML version.

void writeEndDocument(void)
Closes all remaining open start elements and writes a newline.

void writeNamespace(QString namespaceUri, QString prefix = "")

Writes a namespace declaration for namespaceUri with prefix. If prefix is empty, ScriptXmlWriter assigns a unique prefix consisting of the letter 'n' followed by a number. If writeStartElement() or writeEmptyElement() was called, the declaration applies to the current element; otherwise it applies to the next child element.

Note that the prefix xml is both predefined and reserved for http://www.w3.org/XML/1998/namespace, which in turn cannot be bound to any other prefix. The prefix xmlns and its URI http://www.w3.org/2000/xmlns/ are used for the namespace mechanism itself and thus completely forbidden in declarations.

Arguments:

- namespaceUri: The namespace URI.
- prefix: The prefix.

void writeDefaultNamespace(const QString namespaceUri)

Writes a default namespace declaration for namespaceUri. If writeStartElement() or writeEmptyElement() was called, the declaration applies to the current element; otherwise it applies to the next child element.

Note that the namespaces http://www.w3.org/XML/1998/namespace (bound to xmlns) and http://www.w3.org/2000/xmlns/ (bound to xml) by definition cannot be declared as default.

Arguments:

- namespaceUri: The namespace URI.

void writeStartElement(QString name, QString namespaceUri="")

Writes a start element with name, prefixed for the specified namespaceUri. If the namespace has not been declared yet, ScriptXmlWriter will generate a namespace declaration for it. Subsequent calls to writeAttribute() will add attributes to this element.

Arguments:

- name: The name of the element.
- namespaceUri: The namespace URI.

void writeEmptyElement(QString name, QString namespaceUri="")

Writes an empty element with name, prefixed for the specified namespaceUri. If the namespace has not been declared, ScriptXmlWriter will generate a namespace declaration for it. Subsequent calls to writeAttribute() will add attributes to this element.

Arguments:

- name: The name of the element.
- namespaceUri: The namespace URI.

void writeTextElement(QString name, QString text, QString namespaceUri="")

Writes a text element with name, prefixed for the specified namespaceUri, and text. If the namespace has not been declared, ScriptXmlWriter will generate a namespace declaration for it.

This is a convenience function equivalent to:

```
writeStartElement(name, namespaceUri);
writeCharacters(text);
writeEndElement();
```

Arguments:

- name: The name of the element.

- text: The text.
- namespaceUri: The namespace URI.

void writeEndElement(void)
Closes the previous start element.

void writeAttribute(QString name, QString value,QString namespaceUri="")
Writes an attribute with name and value, prefixed for the specified namespaceUri. If the namespace has not been declared yet, ScriptXmlWriter will generate a namespace declaration for it. This function can only be called after writeStartElement() or writeEmptyElement() have been called.
Arguments:

- name: The name of the attribute.
- value: The value of the attribute.
- namespaceUri: The namespace URI.

void writeCDATA(QString text)
Writes text as CDATA section. If text contains the forbidden character sequence "]]>", it is split into different CDATA sections.This function mainly exists for completeness. Normally you should not need use it, because writeCharacters() automatically escapes all non-content characters.

void writeCharacters(QString text)
Writes text. The characters "<", "&", and """ are escaped as entity references "<", "&, and """. To avoid the forbidden sequence "]]>", ">" is also escaped as ">".

void writeComment(QString text)
Writes text as XML comment, where text must not contain the forbidden sequence "--" or end with "-". Note that XML does not provide any way to escape "-" in a comment.

void writeDTD(QString dtd)
Writes a DTD section. The dtd represents the entire doctypedecl production from the XML 1.0 specification.

void writeEntityReference(QString name)
Writes the entity reference name to the internal buffer, as "name;".

void writeProcessingInstruction(QString target, QString data = "")
Writes an XML processing instruction with target and data, where data must not contain the sequence "?>".
Arguments:

- target: The target
- data: The data.

Example scripts
Under the directory exampleScripts several example scripts can be found which demonstrate the 3 different script interfaces of the ScriptCommunicator.