The Bruker SPR Remote API

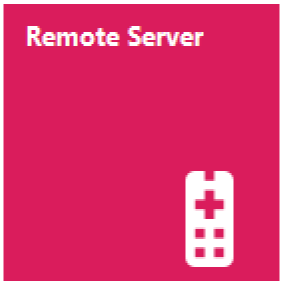
Overview

The Bruker SPR Control Software's remote API allows client frameworks to connect to the Control Software remotely and run its most important operations like starting batch jobs or maintenance operations.
The implementation is based on the Microsoft Windows Communication Foundation (WCF) technology which allows to publish and configure several different endpoints to access the API by just changing the Control Software's configuration file. This allows to write clients using a wide range of different languages, connections, and protocols to talk to the API. In order to keep the API as flexible as possible, it does not support callbacks. Updates about the system's status must be obtained using polling.

As the API is specific for the control software it is highly recommended to have a brief understanding of what you can do with the control software, what the SPR machines of Bruker a capable of, and how they (in principle) work. For that it might be helpful to take a look into the manual of the control software[1].

Remote-Control Mode

Before a client can connect to the API, the Control Software must be switched to the remote-control mode. This mode disables most buttons of the user interface so a user in front of the device cannot interfere with the remote client's commands. The user can only check the status of the device and look at the data. The control software opens a server to which a client can connect. The current user is logged of from the control software when the server is started since the remote-control mode uses a dedicated user account with special permissions.



The remote-control mode is started by clicking the "Remote Server" tile on the Control Software's home screen.

You will get to a new page where you can see a graph with the application's recorded data and a log table showing messages that are exchanged between server and client. Once the server is started, all navigation buttons of the user interface are blocked, and it is not possible to leave the page while the server is running.

Message boxes that are displayed during the regular operation of the Control Software are intercepted and handled by the server to avoid that the software gets blocked while a dialog is shown. Dialogs that allow more than one possible answer can be configured in the file "RemoteServiceDialogResults.xml" which is part of the Control Software's installation.

Simulation Mode

The purpose of the simulation mode is to allow users to plan their assays without using a real machine. It may also be useful for the developers of lab automation frameworks who integrate

[1] The manual can be found in the control software itself, by clicking on "Description" in the right side bar.

the SPR device into their framework. The software is automatically started in simulation mode if no machine is connected.

There is also a feature to simulate errors. It is implemented as batch command ("Throw Error" which you will find in the Method Editor) which can also be used for nonremote testing. The command has no parameters to select. The command will throw an exception when it is executed during a batch job and cause the job to crash with an error message. In order to use the command, you first have to enable it. To do so you have to go to the folder "<installation directory of the control software>/Racks" and edit the file "CommandPrototypesSPR32.xml". At the end of this file you fill find a "BatchMethodElement" with the "<Name>Throw Error</Name>". Remove "<!--" before "<BatchMethodElement …>" and the "-->" after "</BatchMethodElement>" and save the file. Afterwards you have to restart the software. To simulate an error, you have to create a new method with the Method Editor and add the command "Throw Error" to this newly created method. You can also add additional commands before and after the "Throw Error" command and you can add multiple "Throw error" commands.

Endpoint Configuration

The endpoints to which clients can connect are configured in the Control Software's app.config file. Endpoints can be configured for different transfer protocols (TCP or HTTP). Clients can either run locally on the same PC like the Control Software or access the API via a network connection. The base addresses for the endpoints can also be configured. By default, the base address for HTTP endpoints is 9001, resp. 9002 for TCP connections.

Once the remote-control server of the Control Software has been started you can check that it is running by entering the URI http://localhost:9001/bruker-spr/ into a web browser's address field on the computer on which the Control Software is running. You can also do this from a different computer if "localhost" is replaced with the IP address of the Control Software's PC. Of course, the port 9001 must be open to the outside to do this.

You can also use the JSON endpoint to transfer commands directly from the browser to the control software. Some examples:

- http://localhost:9001/bruker-spr/GetOperationMode gets the current operation mode as an integer. The meaning of the value can be picked from the table in the Interface Description section.
- http://localhost:9001/bruker-spr/GetNamesOfMaintenanceProcedures will return a list of names of all maintenance commands that can be run remotely in JSON format.
- http://localhost:9001/bruker-spr/json/RunMaintenanceProcedure?procedureName=Reset%20System starts the maintenance command to reset the device's stepper motors and pumps. This only works if the Control Software is either idle (resp. paused or completed) or in "Reset Required" state (usually after an error).

Client Implementation

Because the server provides several endpoints, there are many ways to implement a client. We do not provide any client applications which are intended for real usage, however there are several demo clients that show how a client application can use the API. You can download the code of these deme clients at https://github.com/BrukerSPR/Bruker-SPR-Remote-API.

C#

The C# implementation can be found in the RemoteAppTestClient project. It is a Windows Forms application where the client object is based on the System.ServiceMode.ClientBase class using a IClientChannel that implements the same data contract interface as the server. The application provides controls to test almost all methods of the API and connects to the service using webHttp or TCP. The connection can be configured in the application's config file.

C++

Two options are given to create a C++ client. Option number 1 depends on Microsoft and its WebServices.dll. This way is described in the ReadMe.txt of the *RemoteAppTestClientConsoleNetTCP* project. The 2nd option is platform independent, uses gSOAP and is described in the ReadMe.txt of the *RemoteAppTestClientConsole* project. Both options do not have a hard-coded connection and their classes (*.c/*.cpp and *.h files) are generated with the described tools (see the specific ReadMe.txt).

Both demo applications are simple console applications providing the same features. Their usage is self-explanatory. After launching them they ask the user to enter a connection URI (or choose the default) and then offer different options which the user can select by entering the corresponding key.

Python, Java Script or other Web Languages

The webHttp endpoint can also be accessed using an arbitrary web browser. The data exchanged is based on JSON and requests can be submitted by just typing into the browser's address bar. It is hence easy to write a client in "web languages" like Python or Java Script. A python example can be found in the *RemoteAppTextClientPython* project. It contains the class Mass1Api which allows to call the remote API functions and a Tools class providing some helper methods. These two classes make it trivial to write a simple script to enqueue operations. An example for such a script is “RemoteAppTestClientPython.py”.

Interface Description

In order to keep the client implementation as flexible as possible, our server does not use callbacks. This means that the server is not capable of contacting the client framework actively if an error has occurred or the state of the software has changed. The client must ask in regular intervals instead (polling). Messages by the control software are queued in a thread-safe queue instead. The arguments and return types are restricted to very fundamental types like numbers, Boolean values, strings, and arrays.

Methods

In Table 1 all available API methods are listed and described. Whenever “method” (or “methods”) is part of the name of the API method or its description, not the or any API method is meant but the methods that can be created and run with the control software (i.e. are part of the runset of the control software). These methods may be also named procedures.

Table 1: API methods

Method Name	Return Type	Arguments	Description
Get information about methods			
GetNamesOfMethods	string[]	-	*Returns* an array of all method names that are loaded in the current runset library.
GetNamesOfMethodsOfAssayType	string[]	string assayType	*Returns* an array containing the names of all methods of the provided assay type that are loaded in the current runset library.
GetAssayTypesOfAllMethods	string[]	-	*Returns* an array containing the assay types of all methods that are loaded in the current runset library.
GetNameOfCurrentMethod	string	-	*Returns* the name of the method that is currently running or selected for running. *Returns* an empty string if there is no method selected.

GetAssayTypeOfCurrentMethod	string	-	*Returns* the assay type of the method that is currently running or selected for running. Returns an empty string if there is no method selected.
GetAssayTypeOfMethod	string	string methodName	*Returns* the assay type of the method with the given name. *Returns* an empty string if there is no method with the given name in the runset library.
Get information about runsets			
GetNamesOfRunsets	string[]	-	*Returns* an array of all runset names that are loaded in the current runset library.
GetNamesOfRunsetsOfAssayType	string[]	string assayType	*Returns* an array containing the names of all methods of the provided assay type that are loaded in the current runset library.
GetAssayTypesOfAllRunsets	string[]	-	*Returns* an array containing the assay types of all runsets that are loaded in the current runset library.
GetNameOfCurrentRunset	string	-	*Returns* the name of the runset that is currently running or selected for running. *Returns* an empty string if there is no runset selected.
GetAssayTypeOfCurrentRunset	string	-	*Returns* the assay type of the runset that is currently running or selected for running. *Returns* an empty string if there is no runset selected.

GetAssayTypeOfRunset	string	string runsetName	*Returns* the assay type of the runset with the given name. *Returns* an empty string if there is no runset with the given name in the runset library.
GetMethodNamesOfRunset	string[]	string runsetName	*Returns* the names of the methods in the runset with the given name.
Prepare and run assays			
SelectMethod	bool	string methodName	Selects the specified method for running. *Returns* 'false' if the method could not be selected. Check for error messages in that case. Possible reasons are that the operation mode doesn't allow to select a method or that the method with the provided name doesn't exist in the runset library.
SelectRunset	bool	string runsetName	Selects the specified runset for running. *Returns* 'false' if the runset could not be selected. Check for error messages in that case. Possible reasons are that the operation mode doesn't allow to select a runset or that the runset with the provided name doesn't exist in the runset library.

CreateRunset	bool	string[] methodNames	Creates a runset that contains the methods specified by the passed array of method names and selects the runset for running. *Returns* 'false' if the runset couldn’t be created. Check for error messages in that case. Possible reasons are that the operation mode doesn't allow to select a runset or that one or more methods with the provided names do not exist in the runset library.
SetSamplePlateId	bool	int methodIndex, string plateId	Assigns an ID to the exchangeable plate of the method with the specified index in the currently selected runset. *Returns* 'false' if the index of the method or if the method with the given index doesn't use a sample plate. Check for errors in this case.
GetSamplePlateId	string	in methodIndex	Gets the ID of the sample plate of the method with the given index in the current runset. *Returns* an empty string if the index is out of range, if there is no runset, no plate, or the plate doesn't have an ID assigned.
GetCurrentSamplePlateId	string	-	Gets the ID of the sample plate that is currently inside the machine. *Returns* an empty string if there is no plate inside the machine or the ID is unknown.

MoveSamplePlateTrayOut	bool	-	Moves the sample plate tray out of the machine so that the plate mover can get or put a plate from it. *Returns* 'false' if this is not possible. This might be the case if there is a runset or maintenance procedure currently running or if a reset is required. Check for errors in this case.
MoveSamplePlateTrayIn	bool	-	Moves the sample plate tray into the housing. *Returns* 'false' if this is not possible. This might be the case if there is a runset or a maintenance procedure is currently running or if a reset is required. Check for errors in this case.
StartSelectedRunset	bool	-	Starts the currently selected runset. *Returns* 'false' if this is not possible. Check for errors in this case. Possible reasons might be that the system is not ready, is on error, or there is no selected runset.
StartSelectedRunsetFrom	bool	int methodIndex	Starts the selected runset starting from the method with the given index. *Returns* 'false' it this is not possible. Check for errors in this case. Possible reasons might be that the system is not ready, is on error, the index of the method is out of range, or no runset is selected.

PauseRunsetAfter	bool	int pauseMode	Pauses the runset that is currently running. The pauseMode argument sets when the runset is paused (1 = after current command, 2 = after current cycle, 3 = after current method). *Returns* 'false' if this is not possible because there is currently no runset running.
ResumeRunset	bool	-	Resumes the paused runset where it was paused. *Returns* 'false' if this is not possible. Check for errors in this case. Reasons might be that the operation mode is different from 'paused' or the device is not ready or on error.
ResetRunset	bool	-	Resets the status of the runset after it was interrupted. *Returns* 'false' if this is not possible because the runset is currently running or if there is no runset to reset. Running the reset method multiple times on the same runset is possible.
AbortScript	bool	-	Aborts the currently running script immediately. That can either be a runset or a maintenance command. *Returns* 'false' if this is not possible because there is nothing running or the running script is already a cancel script (since the device runs a script to clean up the flow cell when interrupted in the middle of an injection).

LeaveStandby	bool	-	Leaves the standby mode. *Returns* 'false' if the machine is not in standby mode when this API method is called.
SetStandbyAfterFinish	bool	bool goToStandby	Sets whether the device should go into standby mode after the runset has finished. Always return true.
GetStandbyAfterFinish	bool	-	Gets whether the device should go into standby mode after the runset has finished. Returns 'true' = yes or 'false' = no.
Maintenance			
GetNamesOfMaintenanceProcedures	string[]	-	*Returns* an array containing the names of all maintenance commands that are available and can be run without user interaction.
RunMaintenanceProcedure	bool	string procedureName	Starts the maintenance procedure with the given name. Returns 'false' if this is not possible. Check for errors in this case. Reasons might be that the device is not ready or there is already something running.
Status			
GetOperationMode	int	-	Gets the current operation mode of the device. The meaning of the return value can be found in Table 2.
IsChipDocked	int	-	Checks whether a chip is docked. *Returns* '1' = docked, '0' = undocked, '-1' = unknown.

IsSamplePlateTrayIn	int	-	Checks whether the sample plate tray is inside the machine. *Returns* '1' = inside, '0' = outside, '-1' = unknown.
HasMessage	bool	-	Checks whether the server's message loop contains at least one new message.
GetMessage	string[]	-	Gets the first message from the server's message queue. A message consists of three strings in an array where the first string is a time stamp formatted like yyyyMMdd-hhmmss.fff, the second gives information about the type of message (Error, Warning, DataSaved, OperationModeChanged, …), and the third string is the message itself. Call this method on a regular base using polling and check the result for null or use the HasNewMessage method first. A complete list of error types can be found in Table 3.
HasErrors	bool	-	Checks whether there are any errors that are not covered by the operation mode.
GetErrors	string[]	-	Gets an array of strings with information about errors like 'no data', 'runset error', ...
HasWarnings	bool	-	Checks whether there are any warnings that are not covered by the operation mode.
GetWarnings	string[]	-	Gets an array of strings with information about warnings like 'temperature unstable', 'runset warning', ...

Operation Modes

Table 2: Operation modes

Numeric Value	Meaning	Description
0	Idle	The device is ready to run a script which can be a runset or a maintenance procedure.
1	Paused	The device is idle but there is a runset selected that was paused during run and might be resumed. The ResetRunset API method has to be called before a new runset can be started. Runing a maintenance command in between is possible, however.
2	Completed	The device is idle and the selected runset has completed. The ResetRunset API method must be called before a new runset can be started.
10	StandBy	The device is currently in standby mode. You have to leave standby to get into the idle, paused, or completed mode.
11	ResetRequired	This state occurs after a stepper motor had an error or if a script was aborted due to an error or by user interaction. The maintenance method 'Reset System' has to be run with the API method RunMaintenanceProcedure to get into idle, paused, or completed mode.
12	MaintenanceRequired	The device is not ready for operation but needs technical maintenance. Currently not used.
13	DoorOpen	The front door of the device is open. It must be closed before the machine can be used.
20	Running	There is currently a script running. This can be either a runset or a maintenance method.

Error Types

Table 3: Error types

Type	Description
Error	A critical error has occurred, current operation cannot proceed
Warning	An uncritical error occurred
StateChanged	The operation mode has changed (see Table 2)

DataCleared	The recorded data were cleared from the application’s memory [Not yet implemented in test version 3.5.0.6]
DataSaved	The recorded data were saved to file [Not yet implemented in test version 3.5.0.6]
RunsetStarted	[Not yet implemented in test version 3.5.0.6]
RunsetCompleted	[Not yet implemented in test version 3.5.0.6]
MethodStarted	[Not yet implemented in test version 3.5.0.6]
MethodCompleted	[Not yet implemented in test version 3.5.0.6]
CycleStarted	[Not yet implemented in test version 3.5.0.6]
CycleCompleted	[Not yet implemented in test version 3.5.0.6]
InjectionStarted	[Not yet implemented in test version 3.5.0.6]
InjectionCompleted	[Not yet implemented in test version 3.5.0.6]
AppMessage	Other message from script [Not yet implemented in test version 3.5.0.6]

Device Geometry

The geometry of the device is shown in Figure 1. All distances are given in millimeters. The plate tray for the sample plate can be moved out to the right side of the device, so the plate can be exchanged by a plate robot controlled by a lab automation framework. There is also space for a plate on the left side of the device, however, the current software does not support to exchange it at runtime.

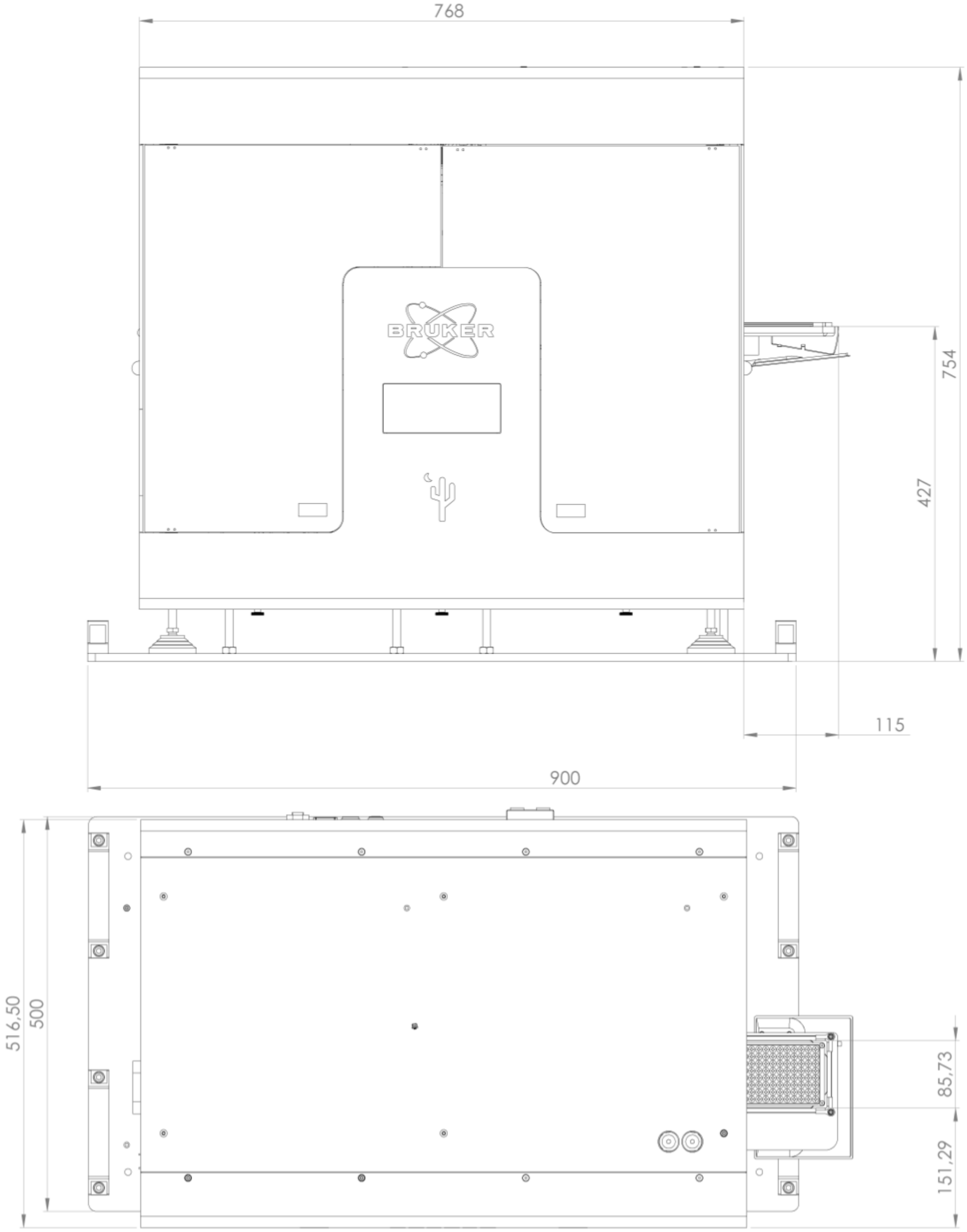


Figure 1: Geometry of the device