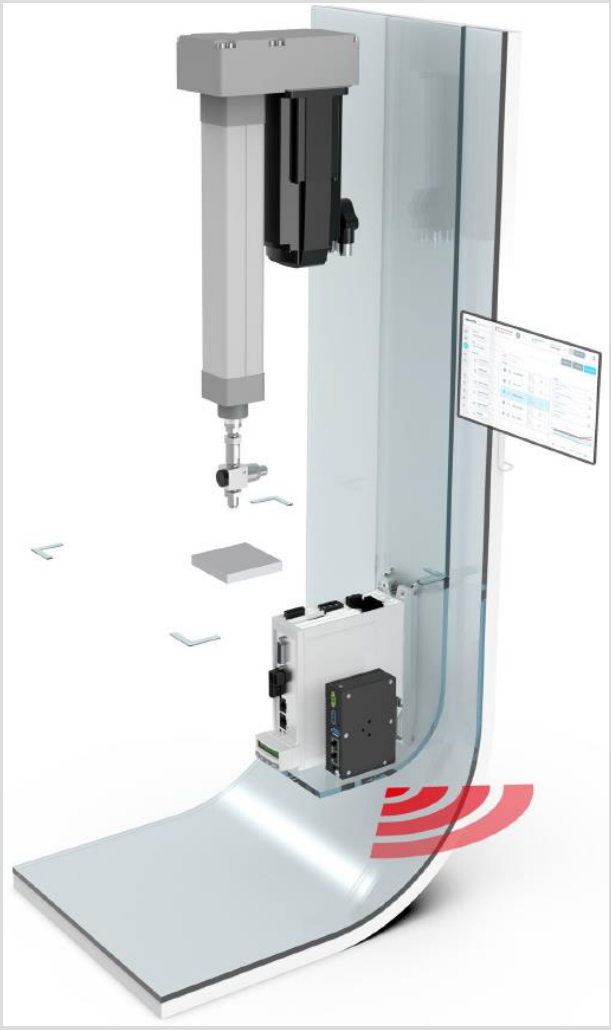
rexroth
A Bosch Company

Smart Function Kit SFK - OPCUA + REST-API

Manual
R320103210 / 2020-10

English



This data has been provided solely for the purpose of product description. Any references to possible uses are provided merely as a convenience and shall be understood as application examples or suggestions.
Catalog data may not be construed as guaranteed characteristics. The information given does not release the user from the obligation of own judgment and verification.
It should be noted that our products are subject to a natural process of aging and wear.

The title page contains an illustration of a sample configuration. The product as delivered can differ from the illustration.

The original instructions are in German.

Any dissemination of the product must include these instructions.

Contents

1 About this documentation

1.1 Validity of the documentation

This documentation applies to the following products:

• Smart Function Kit SFK as described in the Smart Function Kit SFK online catalog.

This documentation is intended for operators, service technicians and system owners.
This documentation contains important information about two communication interfaces of the Smart Function Kit:

- OPC-UA
- REST-API

1.2 Required and supplementary documentation

▶ Only commission the product once you have obtained the system documentation that is marked with the 🕮 and understood and complied with its contents.

Table 1: Required documentation

	Titel	Dokumentnummer	Dokumentart
🕮	Smart Function Kit SFK – instruction	R320103194	Instructions
🕮	Smart Function Kit SFK – Software	R320103208	instructions

The Rexroth documentation is available for download at www.boschrexroth.com/mediadirectory.

2 OPC UA

2.1 Overview

OPC UA (Open Platform Communications United Architecture) is an open communication standard, which is used in industrial environment.
The standard is platform-independent and thus enables manufacturer-independent data exchange.
In order to offer a simple application/integration of this interface, the Smart Function Kit ha a corresponding server implementation (data model), which can be requested by using an OPC client.

2.1.1 Establishing a server connection

To communicate with the Smart Function Kit via OPC UA an OPC Client is required. “UA Expert”, which is offered by Unified Automation GmbH, has proven to be a working client.

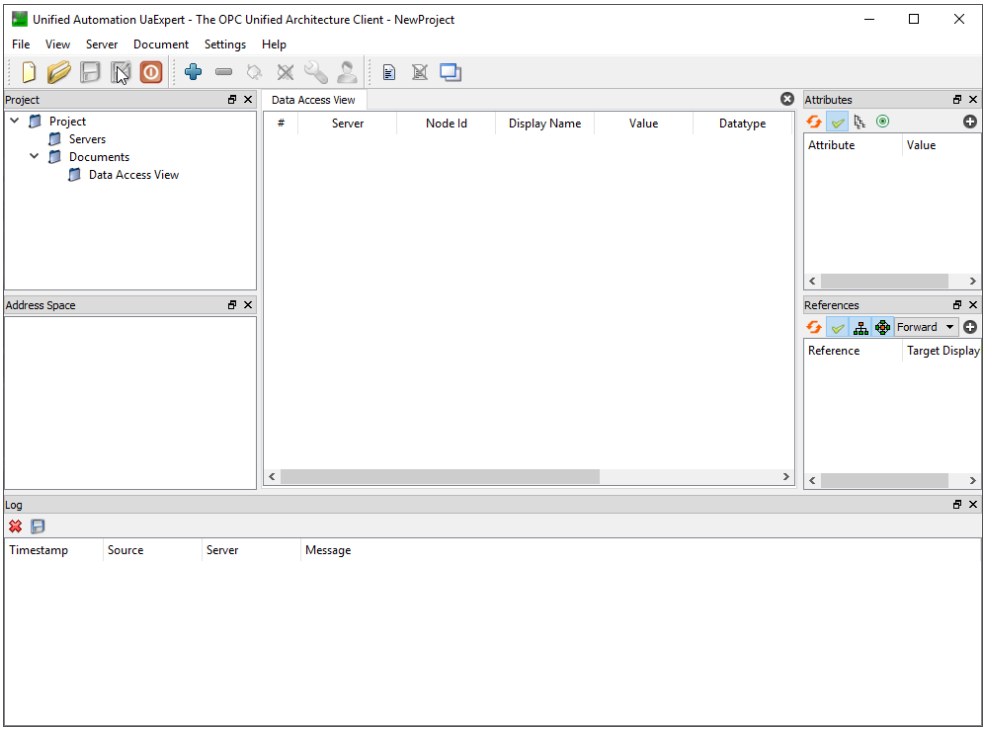


Figure 1: Control surface "UaExpert"

In the OPC Client, the Smart Function Kit must first be added as a server.
There are two protocols for connecting to a server, which differ in the URL of the server:

- Binary protocol
  opc.tcp://<Server>

- Web service
  https://<Server>

The connection is based on the binary protocol. The IP address and port number (port 4334) can be used to search for the server implementation in the network. If all parameters are set correctly, the OPC Client read the data model of the Smart Function Kit and offer the desired interactions.
Connection configuration:

URL: "opc.tcp://192.168.0.1:4334"
No encryption is used for the connection.

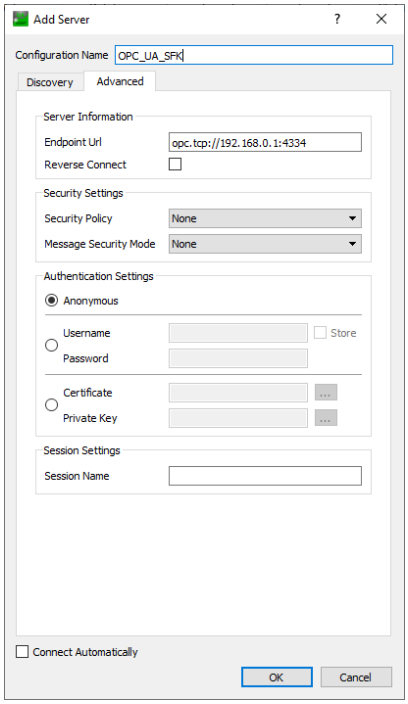


Figure 2: Server configuration

2.2 Data model/ information model

The data of the Smart Function Kit are made available in form of an object (1). The object contains properties (variables) (2) and methods (3), which is comparable to object-oriented programming.

This enables access to a specific variable (e.g. switching frequency of the drive) or the execution of a specific method (e.g. reading out parameters).

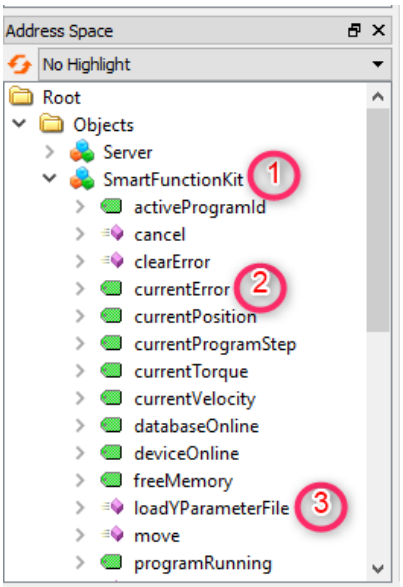


Figure 3: Data model

2.2.1 Methods

The integrated server offers the following methods to interact with the Smart Function Kit.

cancel()

Terminates an active command and aborts it.

clearError()

Acknowledges the current error.

loadYParameterFile()

There are two different types of parameter files. One is used to configure SMC. The other file is used for parameterizing an axis.
The distinction is made in the file naming. (If “Axis” is included in the file name, this file is automatically interpreted for the parameterization of the axis).
The file format of these parameter files is “.scd”.
The two parameter files are used for commissioning and are stored in the “/user” directory on the SD card of the drive.

Eingabe Parameter	Typ	Beschreibung
fileName	String	File with Y parameters without „.scd“

Example:
Load SMC configuration: *Y_Param_System*
Load axis parameterization: *Y_Param_Axis1;*

Y parameter structure:

```
Element                                  | Beispiel
-----------------------------------------+-----------------------------------------
Number of Y-parameter                    Y001
Name of Y-parameter                      Cycle time
Unit of Y-parameter                      µs
Minimum value of Y-parameter             1000
Maximum value of Y-parameter             4000
Actual value of Y-parameter              4000
End of Y-parameter                       |
```

move()
Moves the Smart Function Kit to the specified target position.

Parameter	Type	Description
distanceOrPosition	UINT32	Target position / distance (mm)
relative	BOOLEAN	1 → relative 0 → absolute
velocity	UINT32	Velocity (mm/s)

Example:
move(50, 1, 30);

reboot()
Restarts the entire system.

readHistory()
Reads the command history of the system and lists the last 20 commands/actions that were executed. This is used for diagnostic purposes during the analysis of command sequences.

readIOs()
Reads the inputs and outputs of the system with the corresponding properties and returns the information as a JSON string.
The JSON object contains the following information:

- System name
- Alias name
- Edge (Rising / Falling)
- Type (program / system function)
- Type (input / output)
- State (True / False)
- Inactive (True / False)

readParameter()
Reads a specified S or P parameter and returns the parameter content as a string. It is also possible to read a list parameter.

Parameter	Type	Description
parameterName	String	Parameter ID S-0-xxx or P-0-xxxx

Example:
readParameter(P-0-0001);

readStatus()
Reads the currently active command and its call context. This information can be used to analyze which command or request was sent to the system from which source at what time.
As output, the function returns a string with the corresponding information. This function is used for diagnostic purposes.

readSMCVariable()
Reads a system variable of the underlying technology package SMC and returns the content of the variable as a string.

Parameter	Type	Description
variableName	String	Name of variable

Example:
readSMCVariable(VFR002);

readYParameter()
The method reads a system parameter of the underlying technology packet SMC and returns the content as a string.

Parameter	Type	Description
parameterName	String	Y-Parmeter ID

Example:
readYParameter(Y1007);

setPMOM()
Enables manual switching between parameterization and operating mode.

Parameter	Type	Description
pm	Boolean	1→ Parametermode active 0→ Normal mode active

setProgrammActive()
A program can be loaded or set active. The program ID can be read out in the selection list in the programs area.

Parameter	Type	Description
IdAsName	String	Program ID

Example:
setProgrammActive(5d08fb20c21695);

setReference()
Sets the current reference point of the system to the value set in the extended configuration in parameter S-0-0052.

startProgram()
Starts the currently active program.

startReference()
Starts homing.

tare()
The force sensor is tared to a specific value. Thus, a weight of a tool can be compensated.

Eingabe Parameter	Typ	Beschreibung
offsetInPercent	UINT32	Value to be tared to

Note:
If the force sensor is active, the offset value is the force in [N].
If the motor current is used to determine the force, the percentage torque in relation to the nominal torque of the motor must be used.

writeParameter()
Writes a specified S or P parameter. Many parameters can only be written in parameter mode, so it is recommended to change to parameter mode before writing to the parameter.

Parameter	Type	Description
parameterName	String	Parameter ID S-0-xxx or P-0-xxxx
value	String	Value to write

Example:
writeParameter(P-0-0001);

writeSMCVariable()
Writes a system variable of the underlying technology package SMC.

Parameter	Type	Description
variableName	String	Name of variable
value	String	Value to write

Example:
writeSMCVariable(VFR002, 6.5);

writeYParameter()

The method writes a system parameter of the underlying technology package SMC. Many parameters can only be written in parameter mode, so it is recommend to change to parameter mode before writing to the parameter.

Parameter	Type	Description
parameterName	String	Y-Parameter ID
value	String	Value to write

Example:
writeY parameter(Y1007, 50);

2.2.2 Variables

The following variables can be read and monitored.

activeProgramID <String>

Reads the ProgramID of the actively loaded program.

currentError <Double>

Reads the current error.

Example:
E-Stop activated: *999476*

currentPosition <Double>

Current position relative to the zero point of the system.

currentProgramStep <Double>

Current program step.

currentTorque <Double>

Force applied to the force sensor.

currentVelocity <Double>

Current speed

databaseOnline <Boolean>

Checks whether the system database is online and whether data can be stored there.

deviceOnline <Boolean>
Indicates whether the system is available. This checks whether a TCP connection can be established between PR21 and the motor controller.

freeMemory <String>
Reads the available storage memory of the PR21.

programmRunning <Boolean>
The value 1 symbolizes that the program is currently being processed.

3 REST-API

3.1 Architecture and basics

The Smart Function Kit (SFK) for Pressing includes an Application Programming Interface (API), which makes various system parts accessible from outside. To achieve a deeper understanding in the opportunities of this feature, this chapter describes the functionality of the used representational state transfer (REST) technology and WebSocket communication.

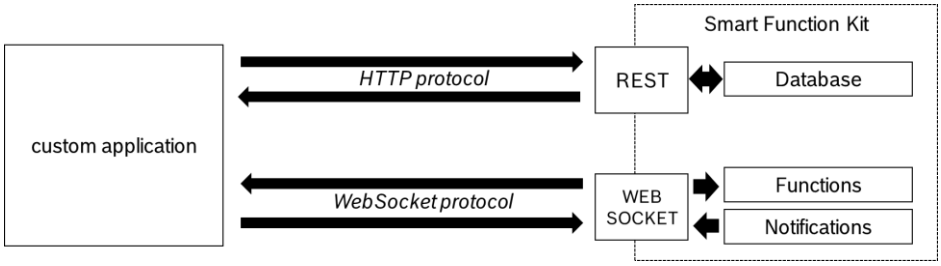


Figure 4: SFK API architecture

In general, the API defines the way for a developer to write a program with requesting services from an operating system (e. g. Linux) or software application (e. g. SFK operating software).
The REST technology defines the way of interaction with the system. It is designed specifically for browser based cloud services interactions and is therefore optimized for a low bandwidth.
A REST API uses the HTTP methodologies defined by the RFC 2616 protocol. Use GET to retrieve a resource; PUT to change the state of or update a resource, which can be an object, file or block; POST to create that resource; and DELETE to remove it.
The REST API of the SPK provides the authentication service and access to the database, e. g. curve data or program configurations.
To get access to the functions, e. g. to start and stop the current program, the WebSocket API of the SFK can be used. This API is based on the server client model and is defined by RFC 6455. The connection type of this protocol is TCP and provides full-duplex communication.
The SFK provides the WebSocket server. Therefore, a WebSocket client has to be established by the custom application.

3.2 API Access Overview

The REST API of the Smart Function Kit provides accessibility to the following data:

- account
- authentication
- configuration
- curves
- programs
- user
- valuation elements
- activity
- commissioning

The WebSocket API provides the following functions:

- Notifications (Status information)
- RPC Calls to SPS implementation (Control of the drive)

A detailed description how to implement specific requests can be found in the end of this document.

3.3 Implementation

There are several different options for an implementation. Various programming languages provide libraries for the implementation of connections based on the http- and WebSocket-protocol (e. g. JavaScript/TypeScript, C#, Java).
The provided examples within this documentation are written in JavaScript.

3.3.1 Connection Check

Before starting programming and testing custom application, it is useful to check whether the connection to the SFK is established. To do so, the User Interface can be viewed by using the browser (firefox, chrome).
By typing in the IP-address (e. g. "*https://192.168.0.1*") of the SFK, the dashboard should be accessible. Additionally, the login credentials can be checked. If there are any issues, please clear the cache of the used browser and load again.

3.3.2 Authentication (POST)

The first step of data extraction always includes an authentication. It can be achieved by sending the user login data via http-request. The response to this request includes a token (cryptic string), which has to be used in further requests headers to authenticate. The authentication type is bearer, which should only be used over HTTPS (SSL). The authentication includes the POST order. This method does always create a new instance, in this case an authentication with the identifier token as a feedback.

Code-Example:

```
function login () {
        const Http = new XMLHttpRequest();
        let ipAddress = '192.168.0.1';      // Enter your SFK IPC IP here
        let password =  'admin';            //Enter your SFK dashboard password here
        let user = {
          username : "admin",
           password : password
        };
        const url= 'https://' + ipAddress + '/api/authenticate';
        Http.open("POST", url);
        Http.setRequestHeader("Content-Type", "application/json");
        Http.setRequestHeader("Accept", "application/json");
        Http.send(JSON.stringify(user));
        Http.onreadystatechange = function() {
        // Login successful, further REST API requests can be made with bearer token
          if(this.readyState == 4 && this.status == 200) {
             const token = JSON.parse(Http.responseText).token;
        // Do something (see following chapters)
             return;
          }
        // Login error
          else if(this.readyState == 4 & this.status != 200) {
             //Handle error
             return;
          }
    }
}
```

3.3.3 Single data collection (GET)

The GET method is used to get data from the database. The authentication token (see Chapter “Authentication”) is required to get access to the data. It is being sent within the request header.
The response/acquired data are natively structured in JSON format. To handle it for further steps in the custom program, the response string can be parsed from JSON to an object.

Code-Example:

```
function GetAccount (token) {

        const Http = new XMLHttpRequest();
       let ipAddress = '192.168.0.1';                    // Enter your SFK IPC IP here
         const url='https://' + ipAddress + '/api/account';

         Http.open("GET", url, false);
         Http.setRequestHeader("Authorization", "Bearer " + token);
         Http.send();

          // Response in JSON format
          let response = JSON.parse(Http.response);
          let ID = response.id;
          let Login = response.login;
          let Language = response.langKey;
          let Created = response.createdDate;
          let LastModified = response.lastModifiedDate;

return;
}
```

3.3.4 Single data removal (DELETE)

The DELETE method is used to remove data from the database. The authentication token (see Chapter “Authentication”) is required to get access to the data. It is being sent within the request header.

Important: The ID required for a program removal is not equal to the program-ID displayed in the dashboard. Get the right ID by querying over the REST API (GET request).

Code-Example:

```
function DeleteProgram(token, program) {
        let ipAddress = '192.168.0.1'; // Enter your SFK IPC IP here

          // program as JSON object (see GET /api/programs/active)
        let prog_id = program.id;
        const url = 'https://' + ipAddress + '/api/programs/' + prog_id;
        console.log('URL: ', url);
        const Http = new XMLHttpRequest();

        Http.open("DELETE", url, false);
        Http.setRequestHeader("Authorization", "Bearer " + token);
        Http.send();

}
```

3.3.5 Single data update (PUT)

The PUT method is used to update data in the SFK database. The authentication token (see Chapter “Authentication”) is required to get access to the data. It is being sent within the request header.
Please note that some PUT procedures require other requests in advance or afterwards to change the data properly. Specific information about the program data model can be found in the API documentation within the help page of the dashboard. To send the update data via the body, it is necessary to define the content type of the request.

Code-Example (Simple):

```
function UpdateTime(token) {
        let ipAddress = '192.168.0.1';                  // Enter your SFK IPC IP here
        var date = new Date().toISOString().substr(0, 19);

        let time = {
          backend : date,
          drive : date
        };

        const url = 'https://' + ipAddress + '/api/configuration/time';
        const Http = new XMLHttpRequest();

        Http.open("PUT", url, false);
        Http.setRequestHeader("Authorization", "Bearer " + token);
        Http.setRequestHeader("Content-Type", "application/json");
        Http.setRequestHeader("Accept", "application/json");
        Http.send(JSON.stringify(time));
}
```

Code-Example (Advanced):

Important: The ID required for a program change is not equal to the program-ID displayed in the dashboard. It can only be accessed by querying over the REST API (GET request).
In case of a program parameter change, the internal “keepLastModifiedDate” parameter has to be set on “false” to operate this procedure.

Note: The updated program has to be (re-)loaded in the PLC to be executed with the changes (via function call or within the dashboard).

```
function UpdateProgram(token, program) {
        let ipAddress = '192.168.0.1';                          // Enter your SFK IPC IP here
        let newStartPosition = 100;
        let prog_id = program.id;

        // Manipulate parameters of JS object
        program.name = "Test_Start_" + newStartPosition + "mm";

        // "Start position" as first step (array!) in program, position parameter 4 (array!)
        program.functions[0].parameters[3].value = newStartPosition;

        // Deactivate setting “keepLastModifiedDate”
        const url1 = 'https://' + ipAddress + '/api/programs?keepLastModifiedDate=false';
        const Http1 = new XMLHttpRequest();

        Http1.open("PUT", url1, false);
        Http1.setRequestHeader("Authorization", "Bearer " + token);
        Http1.setRequestHeader("Content-Type", "application/json");
        Http1.setRequestHeader("Accept", "application/json");
        Http1.send(JSON.stringify(program));

        // Send updated program
        const url2 = 'https://' + ipAddress + '/api/programs/' + prog_id;
        const Http2 = new XMLHttpRequest();

        Http2.open("PUT", url2, false);
        Http2.setRequestHeader("Authorization", "Bearer " + token);
        Http2.setRequestHeader("Content-Type", "application/json");
        Http2.setRequestHeader("Accept", "application/json");
        Http2.send(JSON.stringify(program));

}
```

3.3.6 RPC/Function call

Function calls, e.g. to start or stop the current SFK program, can be sent via WebSocket technology. Therefore, it is required to authenticate first (see Chapter “Authentication”).

Message: { "command": "<commandName>", "params": [], "handle": <Timestamp> }
Answer: { "handle": <Timestamp> }

```
function start() {
        let ipAddress = '192.168.0.1';                    // Enter your SFK IPC IP here
          const url='wss://' + ipAddress + '/websocket/socket/websocket';
        const webSocket = new WebSocket(url);
        webSocket.onerror = (event) => console.log(JSON.stringify(event));
        webSocket.onmessage = (event) => console.log(event.data);
        webSocket.onopen = () => {
        webSocket.send(`{ "command": "startProgram",
                          "params": [false],
                          "handle": ${new Date().getTime()} }`);
        }
}

function stop() {
        let ipAddress = '192.168.0.1';                    // Enter your SFK IPC IP here
        const url='wss://' + ipAddress + '/websocket/socket/websocket';
        const webSocket = new WebSocket(url);
        webSocket.onerror = (event) => console.log(JSON.stringify(event));
        webSocket.onmessage = (event) => console.log(event.data);
        webSocket.onopen = () => {
        webSocket.send(`{ "command": "cancel",
                          "params": [],
                          "handle": ${new Date().getTime()} }`);
        }
}
```

3.3.7 WebSocket Notifications

Notifications are being sent during the link connection („live“ messages excluded) or at value changes. The object type of the messages is JSON. A list of possible notifications is attached in the end of the documentation. It is recommended to filter the messages on the desired topic.

Example:
JSON data model for the force notification: { "subscription": { "force": number } }

```
function notify () {
        let ipAddress = '192.168.0.1';                    // Enter your SFK IPC IP here
        const url='wss://' + ipAddress + '/websocket/socket/websocket';
        const webSocket = new WebSocket(url);

        webSocket.onmessage = function(event) {
          notification = JSON.parse(event.data);
          if (notification.subscription.force != undefined)
          {
            // Do something with the data
                console.log(notification.subscription.force + ' N') ;
          }
}
```

4 API Documentation

4.1 REST

This documentation provides an overview of the SFK REST API paths. For more detailed information, please open the dashboard, go to help and click the link “API Documentation”.

/api/account
GET /api/account
POST /api/account

/api/account/change-password
POST /api/account/change-password

/api/authenticate
POST /api/authenticate

/api/configuration/functions
GET /api/configuration/functions

/api/configuration/ios
GET /api/configuration/ios
PUT /api/configuration/ios

/api/configuration/hardware
GET /api/configuration/hardware
PUT /api/configuration/hardware
DELETE /api/configuration/hardware

/api/configuration/system

GET /api/configuration/system
PUT /api/configuration/system

/api/curves/reference
POST /api/curves/reference

/api/curves
PUT /api/curves
GET /api/curves

/api/curves/{id}
DELETE /api/curves/{id}
GET /api/curves/{id}

/api/curves/{id}/validate
GET /api/curves/{id}/validate

/api/curves/validate/{validationElementId}
GET /api/curves/validate/{validationElementId}

/api/profile-info
GET /api/profile-info

/api/programs/active
GET /api/programs/active

/api/programs/fieldbus/nextId
GET /api/programs/fieldbus/nextId

/api/programs/fieldbus/{id}
GET /api/programs/fieldbus/{id}

/api/programs/{id}
GET /api/programs/{id}
DELETE /api/programs/{id}

/api/programs/{id}/statistics
GET /api/programs/{id}/statistics

/api/programs
GET /api/programs
POST /api/programs
PUT /api/programs

/api/programs/{id}/Smc
GET /api/programs/{id}/Smc
PUT /api/programs/{id}/Smc

/api/programs/transitiongroups
POST /api/programs/transitiongroups

/api/programs/initFunctionCall/{name}
POST /api/programs/initFunctionCall/{name}

/api/programs/{copyId}/copy
POST /api/programs/{copyId}/copy

/api/programs/{id}/export
GET /api/programs/{id}/export

/api/programs/import
POST /api/programs/import

/api/users/authorities
GET /api/users/authorities

/api/users/{login}
GET /api/users/{login}
DELETE /api/users/{login}

/api/users
GET /api/users
POST /api/users
PUT /api/users

/api/validation-elements
POST /api/validation-elements
PUT /api/validation-elements
GET /api/validation-elements

/api/validation-elements/{id}
DELETE /api/validation-elements/{id}

/api/activities
GET /api/activities

4.2 WebSocket

The WebSocket documentation is divided in CallService methods and notifications. These operations can be accessed by the following hierarchy/url:
wss://<ip-address>/websocket/socket/websocket

CallService methods:

clearError
clearError(): *Promise<void>*
Deletes/removes the current error

Move
Move(distanceOrPosition: *number*, relative: *boolean*, velocity: *number*, acceleration: *number*): *Promise<void>*
Move axis
Parameters:

- distanceOrPosition: *number*; The relative distance or the absolute position
- relative: *boolean*; Move relative or absolute
- velocity: *number*; The speed of movement
- acceleration: *number*; The acceleration and deceleration of the move operation

readSMCVariable
readSMCVariable(variableName: *string*): *Promise<any[]>*

See SMC documentation in the SFK software documentation
Parameters:
- variableName: string

setProgramActive
setProgramActive(smcName: *string*): *Promise<void>*
Sets the program as active SMC interpreter program
Parameters:
- smcName: *string*; The SMC identifier

setStepMode
setStepMode(active: *boolean*): *Promise<void>*
Sets or unsets the step mode. When active, the program stops after every EMC step.
Parameters:
- active: *boolean*; If true step mode is switched to active

startProgram
startProgram(): *Promise<void>*
Starts the active SMC program. When step mode is active it proceeds only one EMC step.

startReference
startReference(): *Promise<void>*
Moves the axis to fixed stop and sets its offset

stopProgram
stopProgram(): *Promise<void>*
Stops the running SMC program

writeSMCVariable
writeSMCVariable(variableName: *string*, value: *number* | *boolean*): *Promise<any[]>*
See SMC documentation in the SFK software documentation
Parameters:
- variableName: *string*
- value: *number* | *Boolean*

Notifications:
Notifications are being set in JSON format. These are the different data models:

State database connection: *{ "state": { "database": boolean } }*
State Webconnector: *{ "state": { "webconnector": boolean } }*
State drive connection: *{ "state": { "device": boolean } }*
Current position: *{ "subscription": { "position": number } }*
Current force: *{ "subscription": { "force": number } }*
Current velocity: *{ "subscription": { "velocity": number } }*
State program (true/false): *{ "smc": { "ProgActive": string } }*
Active program: *{ "smc": { "ProgActiveName": string } }*
Active program step: *{ "smc": { "ActEMCStep": string } }*
State step mode (true/false): *{ "smc": { "StatusStepMode": string } }*
Drive-/RPC-error (error-number): *{ "rpc": { "ErrorID": string } }*
Measuring data: *{ "live": { "MeasurementData": [IPoint*] } }*
Curve set: *{ "live": { "ProgStarted": ICurve* } }*
Curve validated: *{ "live": { "ProgValidated": { "curve": ICurve*, "validations":[IValidation*] } } }*

Bosch Rexroth AG
Zum Eisengießer 1
97816 Lohr am Main
Germany
Phone +49 9352 18-1234
Fax +49 9352 18-5678
info@boschrexroth.com
www.boschrexroth.com

Subject to modifications.
Printed in Germany
R320103210 / 2020-10