rexroth
A Bosch Company

Smart Function Kit SFK - OPC UA + Rest-API

R320103210, 2022-01, en

The information in this manual is for product description purposes only. Any information on how to use the product is only an example and a recommendation. Catalog information is not binding. The information in this manual does not release the user from exercising their own judgment and conducting their own tests. Our products are subject to natural wear and aging.

R320103210, 2022-01, en

Table of Contents

1 OPC-UA

1.1 Overview

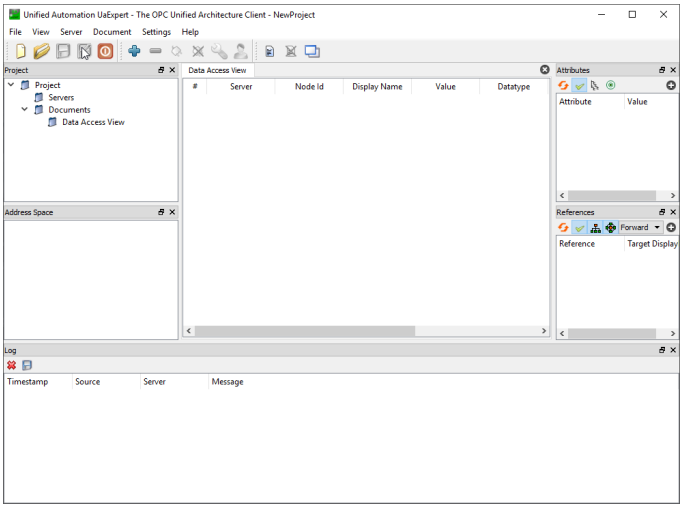
OPC UA (*Open Platform Communications United Architecture*) is an open communication standard used in industrial environments. The standard is independent of platforms and thus allows data exchange regardless of the platform manufacturer.

To ensure easy application and/or integration of this interface, the *Smart Function Kit* is equipped with a corresponding server implementation (data model). In this way the corresponding data model can be requested and used via any OPC client.

1.2 Establishing a server connection

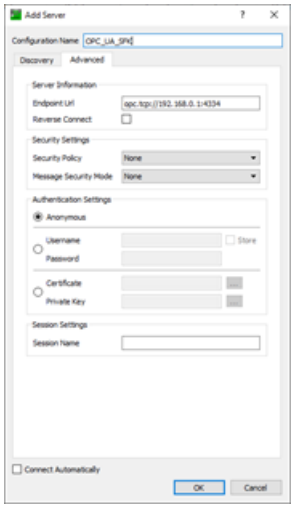
Communication with the Smart Function Kit via OPC UA requires an OPC client.

UA Expert, provided by *Unified Automation GmbH*, has shown to be a functional client.



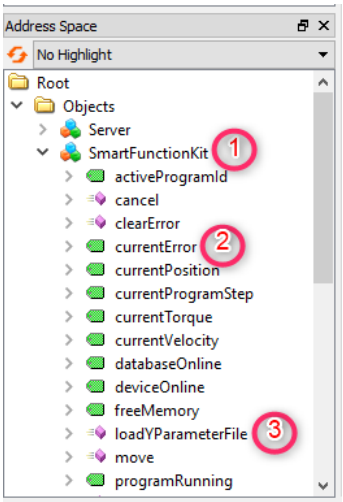
1 UAExpert user interface

Screens/comments	Action steps
Notice Establishing a connection with the Smart Function Kit is done using the binary protocol. There are two protocols for connecting to a server, which can be differentiated by the user by the URL of the server: Binary protocol • *tcp://<Server>* Web service • *https://<Server>*	**First step:** Adding the Smart Function Kit as a server

Screens/comments	Action steps
The required connection configuration is: `URL: "opc.tcp://192.168.0.1:4334"` *Verschlüsselungsart: Keine Verschlüsselung* 	→ Using the IP address and the port number (port 4334), search for the server implementation in the network. ✅ If all parameters are correctly set, the OPC client calls up the data model of the Smart Function Kit and provides the desired interactions.

1 Establishing a server connection

1.3 Data model/information model



2 Structure of the information model

The data of the *Smart Function Kit* are provided in the form of an **object** (1).

Comparable to the object-oriented programming, the object can be assigned various properties and/or **variables** (2) and **methods** (3).

✅ This enables access to a specific variable (e.g. switching frequency of the drive controller) and/or performance of a specific method (e.g. reading a parameter).

1.3.1 Methods

The integrated server provides the following methods for interaction with the Smart Function Kit.

Method	Function	Input parameter	Type	Description
cancel()	Cancels an active command and aborts it.			
clearError()	Deletes all currently pending errors.			

<table>
<tr>
<td>loadYParameterFile()</td>
<td>Here, there are two different types of parameter files. One is used for configuration of SMC and the other for parameterization of an axis. Differentiation is made via the file designation.

(If the name includes the word “axis”, this file is automatically interpreted for parameterization of the axis.)

The file format of both parameter files is “SCD”.

The two parameter file are used for commissioning and after that they are stored in the User directory on the SD card of the drive.

Example:

Loading SMC configuration: Y_Param_System; Loading axis parameterization: Y_Param_Axis1;

Structure of a parameter:
<table>
<tr><th>Element</th><th>Example</th></tr>
<tr><td>Number of Y parameter</td><td>Y1049</td></tr>
<tr><td>Name of Y parameter</td><td>Analog sensor</td></tr>
<tr><td>Unit of Y parameter</td><td>0 / 1</td></tr>
<tr><td>Minimum value of Y parameter</td><td>0</td></tr>
<tr><td>Maximum value of Y parameter</td><td>1</td></tr>
<tr><td>Actual value of Y-parameter</td><td>0: no force sensor
1: force sensor in use</td></tr>
<tr><td>End of Y parameter</td><td>|</td></tr>
</table>
</td>
<td>fileName</td>
<td>String</td>
<td>File with the Y parameters WITHOUT “SCD” suffix</td>
</tr>
</table>

move()	Moves the Smart Function Kit to the specified target position. Example: *move(50, 1, 30,30);*	distanceOrPosition	UINT32	Target position or distance (mm)
		relative	BOOLEAN	1 → relative 0 → absolute
		velocity	UINT32	travel speed (mm/s)
		acceleration	UINT32	Acceleration (mm/s2)
reboot()	Restarts the complete system			
readHistorie()	Reads the command history from the system and lists the last 20 commands/actions that were issued. This is used for diagnostic purposes in the analysis of command processes.			
readIOs()	Reads the set inputs and outputs of the system with their corresponding properties and returns the information as a JSON string. The JSON object contains the following information: • System name • Alias name • Edge (Rising (R)/Falling (F)) • Type (Program (PRG)/System function (SYS)) • Sort (Input (IN)/Output (OUT)) • State (True/False) • Inactive (True/False)			
readParameter()	Reads a specified S or P parameter and feeds back the parameter as a string. Additionally, a list parameter can be read. Example: Switching frequency of the power stage: *readParameter(P-0-0001);*	parameterName	String	Parameter ID in the form of S-0-xxx or-0- xxx

readStatus()	Reads the currently active command and its request context. With this information it can be analyzed which command or request was issued to the system from which source. The output of the function is a string with the corresponding information. The function is used for diagnostic purposes.			
readSMCVariable()	Reads a system variable of the underlying SMC technology package and feeds back the content of the variable as a string. Example: Tolerance for force sensor *readSMCVariable(VFR002);*	variableName	String	Parameter ID in the form of VFRxxx
readYParameter()	The method reads a system parameter of the underlying SMC technology package and feeds back the content as a string. Example: max. acceleration *readYParameter(Y1006);*	parameterName	String	Parameter ID in the form of Yxxxx
setPMOM()	Allows for manual switching between parameterization and operating mode.	pm	Boolean	1 → Parameter mode active 0 → Operating mode active
setProgrammActive()	Created program can be loaded or activated by transmitting the program ID. The program ID can be read from the selection list in the *Programming* area. Example: *setProgrammActive(5d08fb20c21695);*	IdAsName	String	ProgramID
setReference()	Sets the current reference point of the system to the value set for parameter S-0-0052 in the extended configuration.			
startProgram()	Starts the currently active program.			
startReference()	Starts a reference run.			

tare()	the force sensor is tared to any desired value. This allows for compensating for the weight of a tool. **Notice** If the force sensor is active, the force can be specified in [N] as offset. If the motor current is used for determining the force, the torque percentage relative to the nominal torque of the motor must be specified.	offsetInPercent	UINT32	Value to be tared to.
writeParameter()	Writes a specified S or P parameter. Many parameters can only be written in parameter mode. For this reason, before writing the parameter, we recommend changing to parameter mode. Time constant for analog input filter: *writeParameter(P-0-0217,4);*	parameterName	String	Parameter ID in the form of S-0-xxx or-P-0-xxx
		value	String	Value to be written.
writeSMCVariable()	Writes a system variable of the underlying SMC technology package. Example: Tolerance for force sensor *writeSMCVariable(VFR002, 6.5);*	variableName	String	Name of the variable to be read.
		value	String	Value to be written.
writeYParameter()	The method writes a system parameter of the underlying SMC technology package. Many parameters can only be written in parameter mode, so that we recommend changing to parameter mode before writing the parameter. Example: *max. travel range:* *writeYParameter(Y1044, 150);*	parameterName	String	Y parameter ID
		value	String	Value to be written

2 Methods

1.3.2 Variables

The following variables can be read and monitored.

Variable	Function
activeProgramID <String>	Reads the program ID of the currently loaded program.

Variable	Function
currentError <Double>	Displays the currently pending error. Example: E-Stop activated: *999476*
currentPosition <Double>	Current position relative to the zero point of the system.
currentProgramStep <Double>	Current program step of the program being processed
currentForce <Double>	Force applied at the force sensor.
currentVelocity <Double>	Current speed
databaseOnline <Boolean>	Checks whether the system data base is online and if data can be saved there.
deviceOnline <Boolean>	Indicates whether the system is available. It is checked whether a TCP connection can be established between PR21 and the drive controller.
freeMemory <String>	Indicates the available memory capacity of the PR21.
programmRunning <Boolean>	Variables value 1 indicates that the program is currently being executed.

3 Variables

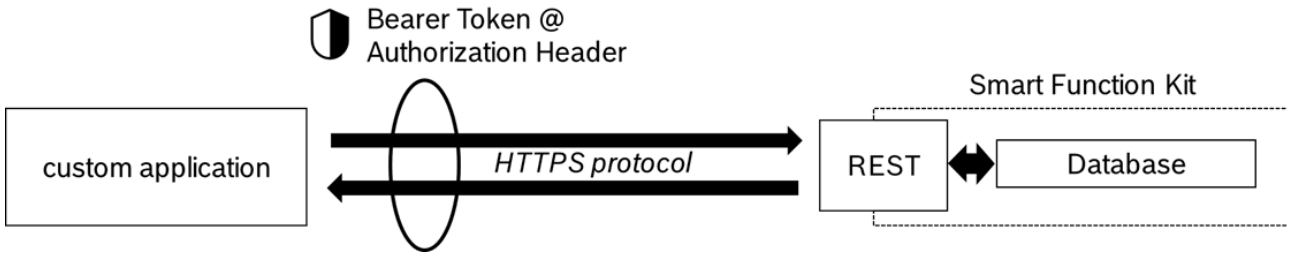
2 REST-API

This documentation provides an overview of the SFK REST paths.

◊ For detailed information, please see the dashboard in the help menu when clicking the link “API documentation”.

2.1 Architecture and basics

The Smart Function Kit (SFK) for press applications contains an application programming interface (API) which allows for external access to various system components. To give you a better understanding of the options this function provides, the functions of the REST technology (Representational State Transfer) used are described in this chapter.



3 SFK API-Architektur

Generally speaking, an API defines how a developer needs to write a program for requesting services from an operating system (e.g. Linux) or a software application (e.g. SFK operating software).

The REST technology defines the interaction method with the system. It is specially intended for browser-based interactions with cloud services and therefore suitable for low bandwidths.

A REST-API uses the HTTP methods defined by the RFC-2616 protocol.

→ Use "GET” to request a resource;

→ Use "PUT” to change the status of a resource (an object, a file, or a block) or to update it;

→ Use "POST" to create a resource;

→ Use "DELETE” to remove a resource.

The REST-API of the SPK provides the authentication service and access to the data base, e.g. curve data or program configurations.

2.2 Overview of API access

The REST-API of the *Smart Function Kit* provides access to the following data:

- Account
- Authentication
- Configuration
- Curves
- Programs
- Users
- Evaluation elements
- Activity
- Commissioning

For detailed descriptions on the implementation of specific requests, please see the end of this document.

2.3 Implementation

There are various options available for implementation. Various programming languages offer libraries for implementation of connections based on the **http** and **WebSocket protocol** (e.g. JavaScript/TypeScript, C#, Java). The examples in this documentation have been written in **JavaScript**.

Connection check

Prior to programming and testing of individual applications, it should be checked that the connection with the SFK has been established. This can be done by calling up the user interface via the browser (Firefox, Chrome). The dashboard should be called up by inputting the IP address (e.g. "https://192.168.0.1“) of the SFK. Additionally, the login data can be checked.

→ In case of problems, clear the cache of the browser used and load it again.

Authentication (POST)

The first step of data extraction is always a form of authentication. It is performed by sending the login data of the user via an http request. The response to this request contains a token (cryptic string) which must be used in the following request headers for authentication. The authentication type is bearer, which should only be used via HTTPS (SSL).

Authentication contains the POST command. This method always generates a new instance as feedback, in this case authentication by the identification token.

Example of a code:

```
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;
                        }
}
}
```

Individual data request (GET)

The GET method is used for requesting data from the data base. The authentication token (see the “Authentication”) is required to obtain access to the data. The authentication token is sent within the context of the request header.

The response/obtained data are natively structured in JSON format. To process in the customer program during the following step, the response string can be parsed from JSON into an object.

Example of a code:

```
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;
}
```

Removal of individual data (DELETE)

The DELETE method is used for removing data from the data base. The authentication token (see the “Authentication”) is required to obtain access to the data. The authentication token is sent within the context of the request header.

Important
The ID required for removing programs is not the same as the program ID shown in the dashboard. You can obtain the correct ID via a request through the REST-API (GET request).

Example of a code:

```
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();
}
```

Updating of individual data (PUT)

The PUT method is used for updating data in the SFK data base. The authentication token (see the “Authentication”) is required to obtain access to the data. The authentication token is sent within the context of the request header.

Please note that for some PUT processes other requests are required beforehand or after in order to ensure correct modification of the data. For specific information on the program data model, please see the API documentation on the help page in the dashboard. To send update data via the body, the content type of the request must be defined.

Example of a code (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));
}
```

Example of a code (advanced)

```
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));

}
```

Important
The ID required for changing programs is not the same as the program ID shown in the dashboard. Access to the correct ID is only possible via a request through the REST-API (GET request).
To be able to perform a change of parameters, the internal "keepLastModifiedDate" parameter must be set to “false”.

Notice
The current program must be uploaded (again) to the PLC to be implemented with the changes performed (via function request or within the dashboard).

RPC/function request

Function requests, e.g. for starting or stopping the current SFK program , can be sent via the WebSocket technology. However, this requires authentication.

Message

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

Response

```
{ "handle": <Timestamp> }
```

RPC/function request

```
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()} }`);
                }
}
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()} }`);
                }
}
```

WebSocket notifications

Notifications are sent during the link connection (except “live” messages) or upon value change.

The object type of messages is JSON. A list of possible notifications is attached to this document. We recommend filtering messages by the desired topic.

Example:

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

WebSocket notifications

```
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') ;
                    }
}
```

2.4 Export of process curves

Process curves can be searched for via the REST-API interface, for example for export to an external data base. Exporting does not require authentication via the bearer token. The structure of the ICurve scheme can be determined via export of a process curve via the web-HMI or the API-Swagger in the web-HMI.

Querying a curve

Query scheme: GET https:// IP-ADDRESS /api/curves/<CURVE-ID>

Response: ICurve with all parameters

Searching the curve data base (Query)

Query scheme: GET https:// IP-ADDRESS /api/curves?<PARAMETER1>&<PARAMETER2>& ....

- Query parameters are linked with "&";
- Parameter scheme: <NAME>=<INPUT-VALUE>

Response: ICurve(s) with all or explicitly selected parameters

Parameter description:

Field name	Mandatory field	Description	Data type	Valid range	Example

page	Yes	Internal page calculation is calculated with "size" (page*size, if page > 0)	number	>=0	page=0 Recommended for all external applications, no offset
size	Yes	Number of displayed data records (Maximum)	number	>1	size=10 Maximum of ten curve sets as response
offset	No	For internal use only			
fields	No	Fields of the response dataset	JSON-Fields designation	All ICurve parameters	fields=id,createdDate,valid,program,maxPosition,maxForce,cycleTime,customId
sort	No	Native Mongo-DB sort request is ignored if simultaneously used with “offset” **Notice** • Depending on the size of the data base, sorting queries may take longer with increasing number of data records. • This must be observed to ensure a stable long-term connection. • The following entries are optimized for long-term stability: _id	Mongo-DB Query	**Possible Fields:** _id, program, valid, status, customId, createdDate, lastModifiedDate, cycleTime, dataRecordingDisabled, validationTime asc/desc	sort={"customId":"desc"} Sorts the data set in a descending order
q	No	Native Mongo-DB filter query	Mongo-DB Query	All ICurve parameters	q={"customId":{"$eq":"34764378691750"}} The query is filtered by curves with a parameter "customId" (serial number) with the value "34764378691750"

Examples

Please try first to access the web-HMI of the SFK to confirm the certificate for the browser. Otherwise, the queries are rejected due to a security error.

Exporting a curve on the basis of the curve ID:

REST Call

```
GET https://IP_ADDRESS/api/curves/6156a3807608d0799f93d88f
```

Exporting the last/latest curve:

REST Call

```
GET https://IP_ADDRESS/api/curves?page=0&size=1&sort={"_id":"desc"}
```

Exporting a curve on the basis of a specific serial number (customId):

REST Call

```
GET https://IP_ADDRESS/api/curves?page=0&size=1&q={"customId":
{"$eq":"34764378691750"}}
```

Exporting a data set (here, the last 5 curves) with specific parameters (here, "id", "valid", "maxPosition", "maxForce", "cycleTime" and "customId"):

REST Call

```
GET https://IP_ADDRESS/api/curves?page=0&size=5&sort={"_id":"desc"}
&fields=id,valid,maxPosition,maxForce,cycleTime,customId
```

Exporting a data set (here, the last 5 curves) without reference curves:

REST Call

```
GET https://IP_ADDRESS/api/curves?page=0&size=5&sort={"_id":"desc"}&q={"reference":
{"$eq":null}}
```

3 API documentation

3.1 REST

This documentation provides an overview of the SFK REST paths. For detailed information, please see the dashboard in the help menu when clicking 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

Table of figures:

Table of tables: