How to connect and use a SICK RFID sensor via SOPAS

by David Schmeding

Prerequisites

This lesson assumes that you want to integrate a SICK sensor device via SOPAS protocol with the Cybus Connectware. To understand the basic concepts of the Connectware, please check out the Technical Overview lesson.
To follow along with the example, you will also need a running instance of the Connectware. If you don't have that, learn How to install the Connectware.
Although we focus on SOPAS here, we will ultimately access all data via MQTT, so you should also be familiar with it. If in doubt, head over to our MQTT Basics lesson.
When coming to the topic of the Rule Engine at the end it can be useful to have basic knowledge of JSON and JSONATA.

Introduction

This article will teach you the integration of a SICK sensor device. In more detail, the following topics are covered:

- Identifying SOPAS commands
- Creating the Commissioning File
- Specifying MQTT mappings
- Establishing a connection via the Connectware Admin UI
- Verifying data in the Connectware Explorer
- Utilizing the Rule Engine

The Commissioning Files used in this lesson are made available in the Example Files Repository on GitHub.

Example setup for this lesson

Our setup for this lesson consists of a SICK RFU620 RFID reading device. RFID stands for „Radio Frequency Identification" and enables reading from (and writing to) small RFID tags using radio frequency. Our SICK RFU620 is connected to an Ethernet network on a static IP address in this lesson referred to as „myIPaddress". We also have the Connectware running in the same network.

About the SOPAS protocol

The protocol used to communicate with SICK sensors is the SOPAS command language which utilizes command strings (telegrams) and comes in two protocol formats: CoLa A (Command Language A) with ASCII

telegram format, and CoLa B with binary telegram format, not covered here. Often, the terms SOPAS and CoLa are used interchangeably, although strictly speaking we will send the SOPAS commands over the CoLa A protocol format. In this lesson we use the CoLa A format only, as this is supported by our example sensor RFU620. (Some SICK sensors only support CoLa A, others only CoLa B, and yet others support both.)
The SICK configuration software SOPAS ET also utilizes the SOPAS protocol to change settings of a device and retrieve data. The telegrams used for the communication can be monitored with the SOPAS ET's integrated terminal emulator. Additional documentation with telegram listing and description of your device can be obtained from SICK either on their website or on request.

Identifying SOPAS commands

For the integration we will need three pieces of information about the SOPAS commands we want to utilize:
1) *Interface type:* This part can be a bit tricky because the terminology used in device documentations may vary. The telegram listing of your device will probably group telegrams in events, methods and variables but sometimes you won't find the term „variable" but only the descriptions „Read"/"Write".
2) *Command name*: Every event, method or variable is addressed using a unique string
3) *Parameters*: In case of variable writing or method calling some parameters may be required

The three interface types mainly have the following purposes:

- Events can be subscribed to and will provide asynchronous messages
- Methods can be called and will be executed by the device
- Variables can be read or written, for example to adjust the configuration of the device

The telegram listing from your device's documentation is the most important source of this information. But for getting a hint of the structure of telegrams, we will take a short look at it.
For example, a command string for the RFU620 that we monitored with SOPAS ET's integrated terminal emulator, could look like this:

```
sMN TAreadTagData +0 0 0 0 0 1
```

The first parameter in this string is the command type which in case of a request can be of the following for us relevant values:

Value	Command type	Interface type
sRN	Read	variable
sWN	Write	variable
sMN	Method call	method
sEN	Event subscription	event

The command type is `sMN` (where `M` stands for „method call“, and `N` for the naming scheme „by name“ as opposed to „by index“). This command name `TAreadTagData` enables us to read data from an RFID tag. Following the command name there are several space-separated parameters for the method call, for example the ID of the tag to read from. In this case we could extract the name `TAreadTagData` and the type `method` from the command string for our Commissioning File but yet don't know the meaning of each parameter so we still have to consult the device's telegram listing.
For this lesson we have identified the following commands of our RFID sensor:

```
| Name          | Type     | Parameters  | Description           |
|---------------|----------|-------------|-----------------------|
| QSinv         | event    | -           | Inventory             |
| MIStartIn     | method   | -           | Start inventory       |
| MIStopIn      | method   | -           | Stop inventory        |
| QSIsRn        | variable | -           | Inventory running     |
| HMISetFbLight | method   | color, mode | Switch feedback light |
```

Writing the Commissioning File

The *Commissioning File* contains all connection and mapping details and is read by the Connectware. To understand the file's anatomy in detail, please consult the Reference docs. To get started, open a text editor and create a new file, e.g. `sopas-example-commissioning-file.yml`. The Commissioning File is in the YAML format, perfectly readable for human and machine! We will now go through the process of defining the required sections for this example:

- Description
- Metadata
- Parameters
- Resources

Description and Metadata

These sections contain more general information about the commissioning file. You can give a short description and add a stack of metadata. Regarding the metadata only the name is required while the rest is optional. We will just use the following set of information for this lesson:

```
description: >

  SICK SOPAS Example Commissioning File for RFU620
  Cybus Learn - How to connect and use a SICK RFID sensor via SOPAS
  https://learn.cybus.io/lessons/how-to-connect-and-use-a-sick-rfid-sensor/

metadata:

  name: SICK SOPAS Example
  version: 1.0.0
  icon: https://www.cybus.io/wp-content/uploads/2019/03/Cybus-logo-Claim-lang.svg
  provider: cybus
  homepage: https://www.cybus.io
```

Parameters

Parameters allow the user to customize Commissioning Files for multiple use cases by referring to them from within the Commissioning File. Each time a Commissioning File is applied or reconfigured in the Connectware, the user is asked to enter custom values for the parameters or to confirm the default values.
parameters:

```
 IP_Address:
    description: IP address of the SICK device
    type: string
    default: mySICKdevice

  Port_Number:
    description: Port on the SICK device. Usually 2111 or 2112.
    type: number
    default: 2112
```

We define the host address details of the SICK RFU620 device as parameters, so they are used as default, but can be customized in case we want to connect to a different device.

Resources

In the resources section we declare every resource that is needed for our application. The first resource we need is a connection to the SICK RFID sensor.

Cybus::Connection

```
resources:

  sopasConnection:
    type: Cybus::Connection
    properties:
      protocol: Sopas
      connection:
        host: !ref IP_Address
        port: !ref Port_Number
```

After giving our resource a name – for the connection it is `sopasConnection` – we define the `type` of the resource and its type-specific `properties`. In case of `Cybus::Connection` we declare which `protocol` and connection parameters we want to use. For details about the different resource types and available protocols, please consult the Reference docs. For the definition of our connection we refer to the earlier declared parameters `IP_Address` and `Port_Number` by using `!ref`.

Cybus::Endpoint

The next resources needed are the endpoints which we supply data to or request from. Those are identified by the command names that we have selected earlier. Let's add each SOPAS command by extending our list of resources with some endpoints.

inventory:

```
  type: Cybus::Endpoint
   properties:
     protocol: Sopas
     connection: !ref sopasConnection
     subscribe:
       name: QSinv
       type: event

  inventoryStart:
    type: Cybus::Endpoint
    properties:
      protocol: Sopas
      connection: !ref sopasConnection
      write:
```

```
      name: MIStartIn
      type: method

inventoryStop:
  type: Cybus::Endpoint
  properties:
    protocol: Sopas
    connection: !ref sopasConnection
    write:
      name: MIStopIn
      type: method

inventoryCheck:
  type: Cybus::Endpoint
  properties:
    protocol: Sopas
    connection: !ref sopasConnection
    read:
      name: QSIsRn
      type: variable

inventoryRunning:
  type: Cybus::Endpoint
  properties:
    protocol: Sopas
    connection: !ref sopasConnection
    subscribe:
      name: QSIsRn
      type: variable

feedbackLight:
  type: Cybus::Endpoint
  properties:
    protocol: Sopas
    connection: !ref sopasConnection
    write:
      name: HMISetFbLight
      type: method
```

Each resource of the type `Cybus::Endpoint` needs a definition of the used protocol and on which connection it is rooted. Here you can easily refer to the previously declared connection by using `!ref` and its name. To define a SOPAS command we need to specify the desired operation as a property which can be `read`, `write` or `subscribe` and among this the command name and its interface type. The available operations are depending on the interface type:

Type	Operation	Result
event	read	n/a
	write	n/a
	subscribe	Subscribes to asynchronous messages
method	read	n/a
	write	Calls a method
	subscribe	Subscribes to method's answers
variable	read	Requests the actual value of the variable
	write	Writes a value to the variable
	subscribe	Subscribes to the results of read-requests

This means our endpoints are now defined as follows:

- `inventory` subscribes to asynchronous messages of `QSinv`
- `inventoryStart` calls the method `MIStartIn`
- `inventoryStop` calls the method `MIStopIn`
- `inventoryCheck` triggers the request of the variable `QSIsRn`
- `inventoryRunning` receives the data from `QSIsRn` requested by `inventoryCheck`
- `feedbackLight` calls the method `HMISetFbLight`

The `accessMode` is not a required property for SOPAS endpoints and is by default set to `0`. But if you want to access specific SOPAS variables for write access which require a higher accessMode than the default 0 (zero), look up the suitable accessMode and its password in the SICK device documentation. Regarding the `accessMode` the Connectware supports the following values:

Value	Name
0	Always (Run)
1	Operator
2	Maintenance
3	Authorized Client
4	Service

Cybus::Mapping

To this point we would already be able to read values from the SICK RFID sensor and monitor them in the Connectware Explorer or on the default MQTT topics related to our service. To achieve a data flow that would satisfy the requirements of our integration purpose, we may need to add a mapping resource to publish the data on topics corresponding to our MQTT topic structure.

mapping:

```
type: Cybus::Mapping
 properties:
   mappings:
     - subscribe:
         endpoint: !ref inventory
       publish:
         topic: 'sick/rfid/inventory'
     - subscribe:
         topic: 'sick/rfid/inventory/start'
       publish:
         endpoint: !ref inventoryStart
     - subscribe:
         topic: 'sick/rfid/inventory/stop'
       publish:
         endpoint: !ref inventoryStop
     - subscribe:
         topic: 'sick/rfid/inventory/check'
       publish:
         endpoint: !ref inventoryCheck
     - subscribe:
         endpoint: !ref inventoryRunning
       publish:
         topic: 'sick/rfid/inventory/running'
     - subscribe:
         topic: 'sick/rfid/light'
       publish:
         endpoint: !ref feedbackLight
```

The mapping defines which endpoint's value is published on which MQTT topic or the other way which MQTT topic will forward commands to which endpoint. In this example we could publish an empty message on topic

`sick/rfid/inventory/start` to start RFID reading and publish an empty message on topic `sick/rfid/inventory/stop` to stop the reading process. While the reading (also referred to as inventory) is running, we continuously receive messages on topic `sick/rfid/inventory` containing the results of the inventory. Similarly when publishing an empty message on `sick/rfid/inventory/check` while having subscribed to `sick/rfid/inventory/running`, we will receive a message indicating if the inventory is running or not.
To provide parameters for variable writing or method calling you have to send them as a space-separated string. For instance, to invoke the method for controlling the integrated feedback light of the device just publish the following MQTT message containing a color and a mode parameter on topic `sick/rfid/light`: `"1 2"`

Installing the Commissioning File

You now have the Commissioning File ready for installation. Head over to the `Services` tab in the Connectware Admin UI and hit the `(+)` button to select and upload the Commissioning File. You will be asked to set values for each member of the section `parameters` or confirm the default values. With a proper written Commissioning File, the confirmation of this dialog will result in the installation of a Service, which manages all the resources we just defined: The SOPAS connection, the endpoints collecting data from the device and the mapping controlling where we can access this data. After enabling this Service you are good to go on and see if everything works out!

Verifying the data

Now that we have a connection established between the SICK RFID sensor and the Connectware, we can go to the `Explorer` tab, where we see a tree structure of our newly created datapoints. Hover an entry and select the eye icon on the right – this activates the live view and you should see data coming in.

Although it is not represented by the Explorer view, on MQTT topics the data is provided in JSON format and applications consuming the data must take care of JSON parsing to pick the desired property. The messages published on MQTT topics contain the keys `"timestamp"` and `"value"` like the following:

```
{
  "timestamp": 1581949802832,
  "value": "sSN QSinv 1 0 8 *noread* 0 0 0 0 0 AC"
}
```

This particular example is an inventory message published on `topic sick/rfid/inventory`. You recognize its `"value"` is a string in the form of the SOPAS protocol containing some values and parameters. This is the original message received from the SICK device which means you still have to parse it according to SOPAS specifications. The Connectware offers a feature that can easily help you along with this task and we will take a look at it in the next step.

Utilizing the Rule Engine

For this lesson we will demonstrate the concept of the Rule Engine with a simpler example than the above. We will look at the answer to a variable read request, which is more intuitively to read since those messages usually only contain the variable's value. For instance, this is the answer to `inventoryCheck` on endpoint `inventoryRunning`:

```
{
  "timestamp": 1581950874186,
  "value": "sRA QSIsRn 1"
}
```

The SOPAS string contains just the command type (`sRA` = read answer), the variable name (`QSIsRn`) and the value `1` indicating that the inventory is running. But essentially we care about the value, because the command type is no further interesting for us and the information about the variable name/meaning is implied in the topic (`sick/rfid/inventory/running`). If this message is so easy to interpret for us, it can't be too complicated for the Connectware – we just need the right tools! And that is where the Rule Engine comes into play.

A rule is used to perform powerful transformations of data messages right inside the Connectware data processing. It enables us to parse, transform or filter messages on the base of simple to define rules, which we can append to endpoint or mapping resource definitions. We will therefore extend the `inventoryRunning` resource in our Commissioning File as follows:

```
inventoryRunning:
   type: Cybus::Endpoint
   properties:
     protocol: Sopas
     connection: !ref sopasConnection
     subscribe:
       name: QSIsRn
       type: variable
     rules:
```

```
    - transform:
        expression: |
          {
            "timestamp": timestamp,
            "value": $number($substringAfter(value, "sRA QSIsRn "))
          }
```

We add the property `rules` and define a rule of the type `transform` by giving the `expression` which is a string in the form of JSONATA language. Rules of the type `transform` are direct transformations, generating for every input message, exactly one output message. The principle of the expression is, that you construct an output JSON object in which you can refer to keys of the input object (the JSON message of this endpoint) and apply a set of functions to modify the content. Please note that the pipe `|` is not part of the expression but a YAML specific indicator for multiline strings, denoting to keep newlines as newlines instead of replacing them with spaces.

We want to keep the form of the input JSON object, so we again define the key `"timestamp"` and reference its value to `timestamp` of the input object to maintain it. We also define the key `"value"` but this time we do some magic utilizing JSONATA functions:

- `$number(arg)` casts the `arg` parameter to a number if possible
- `$substringAfter(str, chars)` returns the substring after the first occurrence of the character sequence `chars` in `str`

The second function will reduce the SOPAS string we refer to with `value` by returning just the characters after the string `"sRA QSIsRn "`, in our case a single digit, which is then cast to a number by the first function. We can apply these functions in this way, because we know that the string before our value will always look the same. For more information about JSONATA and details about the functions see https://docs.jsonata.org/overview.html.

After installing the new Service with the modified Commissioning File, we do now receive the following answer to an `inventoryCheck` request (while inventory is running):

```
{
  "timestamp": 1581956395025,
  "value": "1"
}
```

This value is now ready-to-use and applications working with this data do not have to care about any SOPAS parsing!

The Connectware supports some more types of rules. To give you a hint, what might be possible, here is a quick overview:

- `parse` – parse any non JSON data to JSON
- `transform` – transform payloads into new structure
- `filter` – break the message flow, if the expression evaluates to a false value
- `setContextVars` – modify context variables
- `cov` – Change-on-Value filter that only forwards data when it has changed
- `burst` – burst-mode that allows aggregation of many messages
- `stash` – stash intermediate states of messages for later reference
-

For more information about rules and the Rule Engine check out the Connectware Docs.

Summary

We started this lesson with a few SOPAS basics and learned which information about the SOPAS interface is required to define the Commissioning File for the integration of our SICK RFU620. Then we created the Commissioning File and specified the SOPAS connection, the endpoints and the MQTT mapping. Utilizing the Commissioning File we installed a Service managing those resources in the Connectware and monitored the data of our device in the Explorer of the Admin UI. And in the end we had a quick look at possibilities of preprocessing data using the Rule Engine to get ready-to-use values for our application.

Where to go from here

The Connectware offers powerful features to build and deploy applications for gathering, filtering, forwarding, monitoring, displaying, buffering, and all kinds of processing data… you could also build a dashboard for instance? For guides checkout more of Cybus Learn.

Cybus is a specialist for secure IIoT Edge software, headquartered in Germany. Cybus Connectware serves smart factories as a universal Edge and DevOps hub. Machine builders and providers of IIoT services use the Cybus Connectware as a software-based gateway. As early as 2017, Cybus published the first secure industrial connector for machine data according to today's DIN SPEC 27070 standard. Industry analyst Gartner named Cybus a worldwide "Cool Vendor". Today, the company counts medium-sized and large companies from numerous industrial sectors such as mechanical engineering, automotive and aviation among its customers.

Cybus GmbH · Osterstraße 124 · 20255 Hamburg · Germany · www.cybus.io · hello@cybus.io · (+49) 40 228 58 68 51