CLOUD OF THINGS

Cloud Fieldbus

Version 1.4
Date 05.02.2019

ERLEBEN, WAS VERBINDET.

LIST OF CONTENT

1. INTRODUCTION

Cloud Fieldbus (CFB, working name) enables you to connect any fieldbus device to a remote, central management system – Cloud of Things. This connection can be done within minutes and at minimal cost and provides high levels of security and reliability. Connected devices can be completely managed from Cloud of Things including data collection, visualization, fault management and remote control. The following sections describe the main properties and the architecture of CFB, starting with a motivation and an overview of the architecture. Then, the data model for fieldbus devices is discussed along with its mapping to JSON. The mapping to SmartREST is discussed and finally, the user interface is shown.

Note: Fieldbus in this context refers to protocols such as Modbus, M-Bus and Profibus.

2. MOTIVATION

CFB solves five fundamental challenges of fieldbus devices.

Challenge: Remote Management

Fieldbus protocols and gateways are, as such, not easily usable in wide-area networks. They also cannot be used outside of physically protected domains due to lack of security mechanisms. To solve this issue, CFB translates fieldbus protocols into a protocol that works without configuration over any wide-area network and provides confidentiality, authentication and authorization.

Challenge: Fieldbus simplicity

One of the key success factors of fieldbuses are their simplicity and their low implementation cost. However, the protocols as such do not provide enough information to actually interpret the data exchanged over the buses and manage the devices. For example, Modbus does not specify that a particular register should be regularly queried to build a time-series of meter readings. To solve this issue, CFB provides additional metadata that allows a user with basic technical competence to configure a device for remote management.

Challenge: Cost

Today, most existing fieldbus gateways come at a relatively high cost and often do not provide remote management functionality. This results in higher implementation cost, rendering many simpler use cases unprofitable. To solve this issue, CFB assumes very low system requirements on the terminals that are connecting the fieldbus devices to the remote management system, essentially just minimally changing the protocol logic but leaving all interpretation to the remote management system.

Challenge: Fieldbus diversity

Fieldbus devices are very diverse. There are thousands of different types of devices which range from very simple switches to block heatings. It is not economic to support fieldbus devices in a stovepipe implementation. Rather, CFB offers a metadata-driven approach to manage the devices.

Challenge: Openness

With the diversity of the devices, there is also a large diversity of applications for the devices. Again, it is not efficient to provide stovepipe implementation of applications that just support a number of devices – which is today mostly the case. CFB in contrast leverages the power of the Cloud of Things platform to interface fieldbus devices with an ecosystem of IoT applications.

3. ARCHITECTURE OVERVIEW

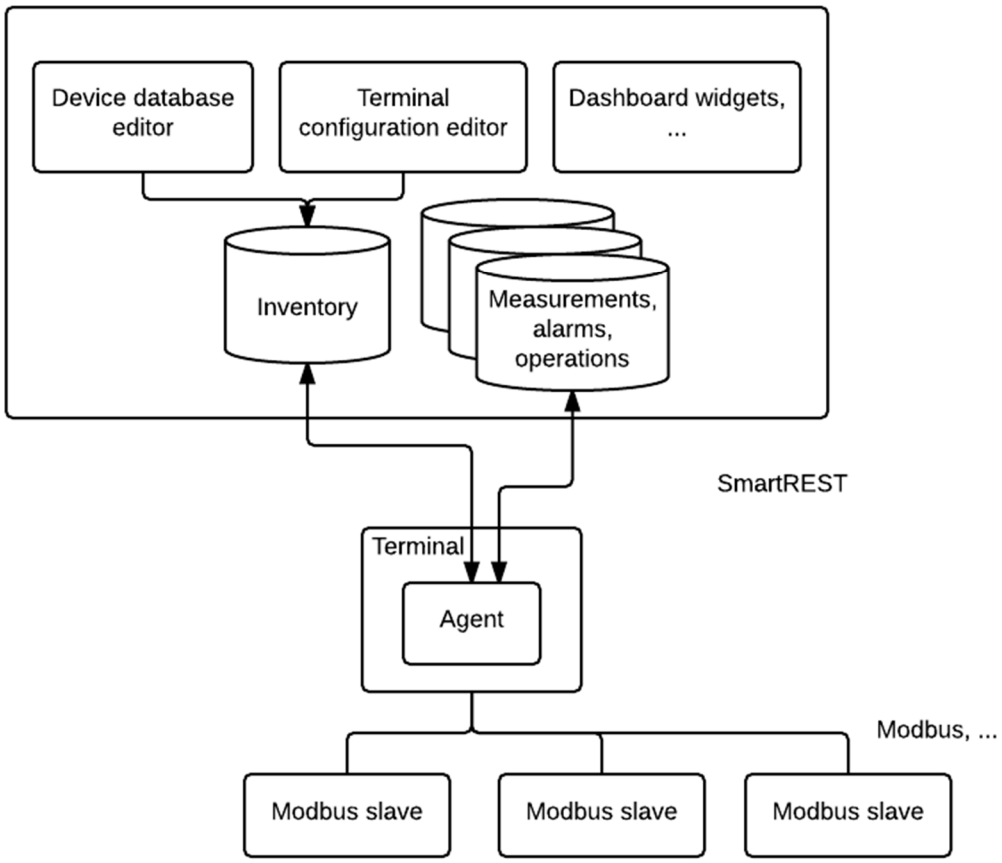
The picture below illustrates the architecture of the platform. Fieldbus devices are connected through a terminal to Cloud of Things. The terminal runs a Cloud of Things agent. On the "device-facing" side, the agent speaks fieldbus protocols to operate the devices, i.e., to read parameters from the devices and to write parameters to

the devices. On the "network-facing" side, the agent speaks the Cloud of Things SmartREST protocol to send parameters to Cloud of Things and receive parameters from Cloud of Things.

To understand what devices with what potential parameters are connected to the terminal, the terminal reads configuration information from Cloud of Things' inventory. The configuration information also enables Cloud of Things to map raw parameters to meaningful concepts such as alarms, measurements and status updates in Cloud of Things.

The configuration information is edited by two plugins in the Cloud of Things web application. The first plugin is the "Device Database Editor", which specifies the available types of devices, what parameters they have and how to map these parameters to Cloud of Things concepts. The second plugin is the "Terminal Configuration Editor", which specifies what devices are connected on what address to a terminal in a particular plant.

Data from fieldbus can be visualized using standard Cloud of Things functionality, such as alarm lists, measurement graphs and dashboards. In addition to the existing visualization functionality, dashboard plugins are provided to show and change states and configuration information in fieldbus devices.



4. SUPPORTED PROTOCOLS

Current Cloud Fieldbus implementation supports three protocols

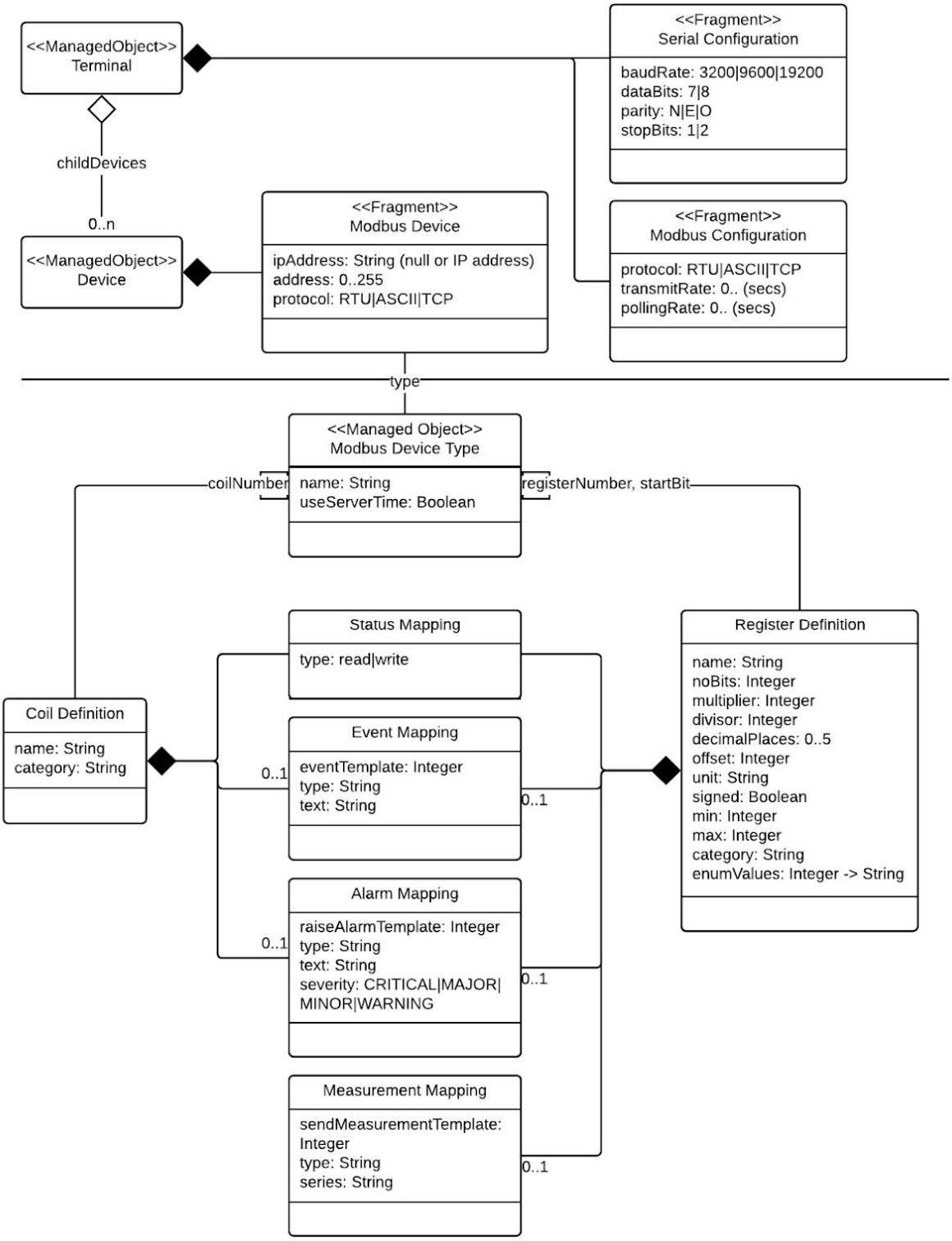
- ModBus
- CanBus
- OPC-UA
- Profibus DP

- CANopen

5. THE DOMAIN MODEL

5.1. Overview

The diagram below illustrates the general domain model of CFB. The generic model is closely based on the Modbus protocol hence the usage of modbus specific terminology. Other protocols may add or omit functionality based on.



Please note that all values, e.g. baud rate are example values.

The upper half of the diagram shows the instance level with a terminal and its connected fieldbus devices:

- A terminal is represented as a managed object in Cloud of Things.
- A terminal contains a Fieldbus configuration fragment, defining
  - Connectivity settings specific for the supported protocols by the terminal
  - The polling rate, i.e., the rate at which the terminal sends requests to the Fieldbus devices to determine changes in registers and coils.
  - The transfer rate, i.e., the rate at which the terminal sends regular measurements to Cloud of Things.
- If a serial protocol is used (RTU, ASCII), the terminal contains also a serial configuration fragment defining the serial parameters in use.
- The connected Fieldbus devices are represented as child devices of the terminal. Each connected Fieldbus device contains its address or protocol specific identifier and a reference to the device type. If TCP is used as transport protocol, an IP address needs to be specified as well.

The lower half of the diagram shows the type/metadata level with the types of devices that can be configured:

- A Fieldbus device type represents a certain make and version of a device, e.g., a certain model of a pump by a certain manufacturer. All devices of this type have the same parameters. A device type is represented by a name, which should be qualified enough to determine manufacturer, model and potentially version of the device.
- A device type has a set of coil definitions and a set of register definitions. Both coil and register definitions have a human-readable name associated with them for visualization in the user interface ("name").
  - A coil definition represents a readable and potentially writable bit in Fieldbus. For example, a coil definition may be named "Operating mode".
  - A register definition represents a readable and potentially writable number in Fieldbus. For example, a register definition may be named "Temperature".
  - For some protocols (here Modbus) there are read only versions called "discrete input" and "input register"
- A coil and a register are physically addressed on Fieldbus protocols using a register or coil number ("number"). Note: We are specifically using the human readable addressing number here, as opposed to index or (storage) address.
- Physical registers in Fieldbus are values depending on the protocol used. In some implementations, additional information is modelled on top of these values. Hence, we distinguish between physical registers, which are the values that are read from the fieldbus device, and logical registers, which are extracted from the physical registers using a start bit and a number of bits. For example, a single physical 16 bit Modbus register may be modelled as two 8 bit logical integer values (start bit = 0 and start bit = 8 with number of bits = 8). Hence:

A register definition contains two additional values "startBit" and "noBits" (number of bits) that determine the subset of bits to read from the register.

  - Register definitions can be repeated for the same physical register number, provided "startBit" is different.
  - Bits are numbered according to the Modbus specification with MSB leftmost.
  - Logical registers should not overlap.
- The value of a logical register can be transformed by the terminal before it is sent to Cloud of Things. To support floating point numbers on terminals without floating point processing capabilities, the value
  - Is first multiplied by "multiplier".
  - Then divided by "divisor".
  - Then offset by "offset" (Optional, only available for CAN protocol currently)
  - And finally, the decimal point is shifted by "decimalPlaces".
- Each logical register can be assigned a unit ("unit"), such as "C" or "m" that is shown in the user interface along with the value.

- Each logical register can represent a signed or unsigned value through the “signed” flag (using the usual two’s complement interpretation).
- A minimum (“min”) and maximum (“max”) can be defined for a logical register. These numbers are used for validation when writing registers to devices.
- The “category” of a coil or register is used for grouping coils or registers into sections in the visualization (e.g., “setpoint parameters”, “actual parameters”, “compressor parameters”).
- "Status Mapping", "Alarm Mapping", “Event Mapping” and "Measurement Mapping" define what to do with coil and register definitions in terms of sending the data to Cloud of Things.
    - An alarm mapping defines if a change in the status of a coil or register should trigger an alarm to be sent to Cloud of Things. For this, the alarm type, text and severity are required. For logical registers, the alarm is triggered when the value of the register is not zero (to support bits modelled on top of physical registers). The Alarm is cleared again when the value returns to zero.
    - An event mapping simply logs changes to coils and registers as event in Cloud of Things with a fixed structure.
    - A measurement mapping describes how to regularly send a register value as measurement to Cloud of Things so that it can be shown in a graph or similar. For this, the measurement type, series and unit are required.
- A status mapping defines how to show a coil or register as status in the inventory. It also defines how to update a coil or register from Cloud of Things (type "write"). For visualization, a mapping of values to text strings can be provided (“enumValues”). For example, if you map “0” to “Off” and “1” to “On”, these texts are shown in a switch in the user interface instead. If more than two values are provided, just the text is shown (resp. a drop-down list for writable registers).
- The flag “useServerTime” defines if the terminal will send timestamps in its measurements, alarms and events or not. This only applies to the generated SmartREST templates generated.
- Generally, alarms and events configured in their respective mappings should be sent to Cloud of Things whenever the status changes. Status updates and measurements configured should only be sent to the platform in the transmit interval configured for the terminal

The translation of the UML model to JSON as used in the Cloud of Things APIs is shown by example.

5.2. MODBUS CONFIGURATION

The cloud fieldbus model is originally based on the modbus protocol. All other protocol integrations are closely related to the original modbus model. The Modbus integration can be considered as reference integration. Differences and changes other protocol integrations introduce are explained in the protocol specific sections.

5.2.1. DEVICE TYPE MANAGED OBJECT

```
{
  "name":"WiloVarioCOR_1.0",
  "type":"c8y_ModbusDeviceType",
  "fieldbusType": "modbus",
  "c8y_Coils":[
    {
      "number": 1,
      "name": "Operating mode",
      "input": false,
      "statusMapping":{
        "status": "write",
      }
      "enumValues":{
        "0": "Manual",
        "1": "Automatic"
      }
    },
    {
      "number": 2,
```

```
      "name":"Power",
      "input": false,
      "statusMapping":{
        "status":"read"
      }
      "enumValues":{
        "0": "Off",
        "1": "On"
       }
    },
    {
      "number":3,
      "name":"Door",
      "input": false,
      "alarmMapping":{
        "text": "Door open",
        "raiseAlarmTemplate": 300,
        "severity": "MAJOR",
        "type": "c8y_DoorOpen"
      }
    }
  ],
  "c8y_Registers":[
    {
      "number": 1,
      "input": false,
      "startBit": 0,
      "noBits": 16,
      "name": "Operating temperature",
      "multiplier": 1,
      "divisor": 1,
      "decimalPlaces": 2,
      "unit": "C",
      "signed": true,
      "min": -30,
      "max": 100,
      "category": "Actual values",
      "measurementMapping":{
        "series": "T",
        "sendMeasurementTemplate": 301,
        "type": "c8y_TemperatureMeasurement"
      }
    },
    {
      "number": 2,
      "input": false,
      "startBit": 0,
      "noBits": 3,
      "name": "Sample enum",
      "multiplier": 1,
      "divisor": 1,
      "decimalPlaces": 0,
      "unit": "",
      "signed": false,
      "min": 0,
      "max": 7,
      "category": "Mode",
      "statusMapping":{
        "status":"write"
      }
      "enumValues":{
```

```
        "0": "Off",
        "1": "Mode1",
        "2": "Mode2",
        "3": "Mode3",
        "4": "Mode4",
        "5": "Mode5",
        "6": "Mode6",
        "7": "Mode7",
      }
    },
    {
      "number": 2,
      "input": false,
      "startBit": 3,
      "noBits": 1,
      "name": "Sample bit",
       "multiplier": 1,
      "divisor": 1,
      "decimalPlaces": 0,
      "unit": "",
      "signed": false,
      "min": 0,
      "max": 1,
      "category": "Mode",
      "statusMapping":{
        "Status":"write",
      },
      "eventMapping": {
         "eventTemplate": 304,
         "type": "c8y_SampleBitEvent",
         "text": "Sample bit changed"
      },
      "enumValues":{
        "0": "Off",
        "1": "On"
      }
    }
  ],
}
```

5.2.2. TERMINAL MANAGED OBJECT

```
{
  "name":"Terminal",
  "type":"myType",
  "c8y_IsDevice":{},
  "c8y_SerialConfiguration":{
    "parity":"N",
    "stopBits":2,
    "dataBits":8,
    "baudRate":9600
  },
  "com_cumulocity_model_Agent":{},
  "c8y_ModbusConfiguration":{
  "transmitRate":60,
    "pollingRate":30
  }
}
```

5.2.3. DEVICE MANAGED OBJECT

```
{
  "name":"Pump",
  "c8y_ModbusDevice":{
    "address":1,
    "protocol":TCP,
    "ipAddress":"192.168.0.2",
    "type":"/inventory/managedObjects/14400"
  }
}
```

Note that devices connected through a serial line do not contain an "ipAddress" parameter. The parameter is put here just for illustration.

5.3. INITIALIZATION

We assume that the standard device registration and inventory upload process is carried out from the terminal as described in **Cloud of Things RestApi Guide.**
This process can be incrementally implemented. E.g., for demonstration or pilot purposes, the terminal could be manually pre-registered in Cloud of Things.
Please note that while we are using JSON format to describe the requests and responses between the terminal and Cloud of Things, the terminal can at any time define own SmartREST templates to simplify parsing and processing.
After the initial setup, the terminal will wait for operations to be sent from Cloud of Things which it processes. The operations are processed according to the standard Cloud of Things operations process (pending, executing, successful/failed, see above documentation and the SmartREST templates in the appendix). If parameters on the terminal are changed, the terminal sends corresponding update messages to the Cloud of Things inventory.
If no Fieldbus devices so far connected, there are operations the terminal can receive. These operations are used for configuring the communication parameters.

- c8y_SerialConfiguration: The terminal receives serial configuration parameters (only for serial protocol variants)

```
{
  "c8y_SerialConfiguration":{
    "baudRate": 9600,
    "stopBits": 1,
    "parity": "N",
    "dataBits": 8
  }
}
```

- c8y_ModbusConfiguration: The terminal receives regular polling and transmit rate.

```
{
  "c8y_ModbusConfiguration":{
    "protocol": "RTU",
    "transmitRate": 60,
    "pollingRate": 30
  }
}
```

- c8y_ModbusDevice: The terminal is notified that a new Modbus device is connected.

```
{
  "c8y_ModbusDevice":{
    "id": "48342012" ,
    "name": "pump_v1" ,
    "address": 1,
    "ipAddress": "192.168.1.11" ,
    "type": "/inventory/managedObjects/30939098",
    "protocol": "TCP"
  }
}
```

(IP address is only included for Modbus TCP variant)

The terminal may at any time use own SmartREST templates to simplify parsing and processing of these requests and responses.

5.4. DEVICE TYPES VERSIONING

Device types are versioned indicating lowest fieldbus version that can handle all features used in the model, e.g. if device type contains no registers using `noBits` larger than 16 and no `littleEndian` option, it will be marked as version 4, but if there is at least one register using more than 16 bits or using `littleEndian` option, model will be marked as version 5. Version is indicated by root property `fieldbusVersion` (see example below).

Example device type with version 5 features:

```
{
      "name": "Sample version 5 Modbus model",
      "fieldbusType": "modbus",
      "fieldbusVersion": 5,
      "c8y_Coils": [
         (...)
      ],
      "c8y_Registers": [
        {
            "signed": false,
            "measurementMapping": {
            "series": "Pressure",
            "type": "c8y_Process"
            },
            "number": 1,
            "divisor": 10,
            "multiplier": 1,
            "decimalPlaces": 0,
            "unit": "mbar",
            "startBit": 0,
            "input": true,
            "category": "Process data",
            "statusMapping": {
              "status": "read"
            },
            "name": "Pressure",
            "noBits": 32,
            "littleEndian": true
        },
        (...)
      ]
   }
```

5.5. FIELDBUS DEVICE VERSIONING

Fieldbus devices can indicate highest fieldbus version of device types they can handle by setting `c8y_ModbusConfiguration.maxFieldbusVersion` (see example below). Supported fieldbus version is displayed in Modbus tab of a device. When adding child devices, the list of selectable device types is filtered by supported version so device types with version 5 are unavailable for gateways handling version 4.

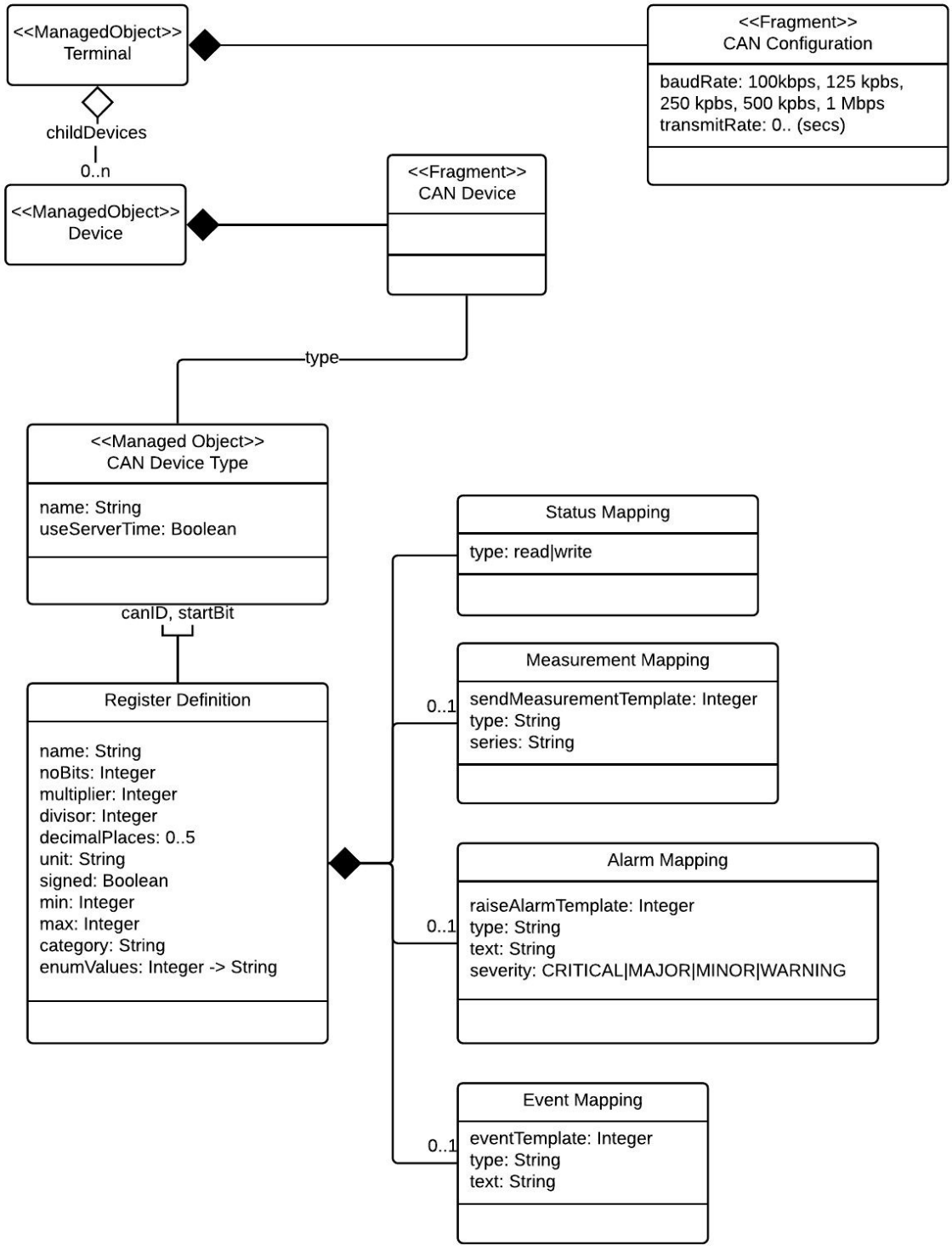
```
{
      "name": "ModbusDevice",
      "type": "POSIX Agent",
      "c8y_IsDevice": {},
      "com_cumulocity_model_Agent": {},
      "c8y_ModbusConfiguration": {
            "pollingRate": 15,
            "protocol": "TCP",
            "transmitRate": 60,
            "maxFieldbusVersion": 5
      }
}
```

Due to differences in Fieldbus protocols there are also slight differences in the mapping of their data into the Cloud Fieldbus domain model.

5.6. CAN

- In CAN protocol CAN messages are mapped to holding registers. Input registers and both types of coils are not used.
- CAN message ID are entered and delivered hexadecimal numbers.
- Polling rate is not used in CAN configuration because the gateway will simply listen to the CAN bus. There is no distinct polling for data.
- There is no concept of a device or registers in CAN protocol. The logical register concept from the generic CFB domain model is still used but in a more abstract form.
- A logical register is represented by a CAN message (or parts of it) identified purely by its message ID.

5.6.1. CAN SPECIFIC DOMAIN MODEL



5.6.2. TERMINAL MANAGED OBJECT

```
{
  "name":"Terminal",
  "type":"myType",
  "c8y_IsDevice":{},
  "com_cumulocity_model_Agent":{},
  "c8y_CanConfiguration":{
    "baudRate": 125000,
    "transmitRate": 60
  },

  c8y_SupportedOperations: [
      "c8y_CanDevice"
  ]
}
```

5.6.3. DEVICE TYPE MANAGED OBJECT

```
{
  "name": "Sweeper_v1",
  "type": "c8y_ModbusDeviceType",
  "c8y_useServerTime": true,
  "fieldbusType": "canbus",
  "c8y_Coils": [],
  "c8y_Registers": [
    {
      "min": null,
      "max": null,
      "signed": false,
      "measurementMapping": {
        "series": "Sweeper",
        "sendMeasurementTemplate": 300,
        "type": "c8y_WorkingHours"
      },
      "number": "0x100",
      "divisor": 1,
      "multiplier": 5,
      "decimalPlaces": 2,
      "unit": "h",
      "startBit": 0,
      "input": false,
      "name": "Working hours sweeper",
      "statusMapping": {
        "status": "read"
      },
      "noBits": 32,
      "offset": 0
    },
    {
      "min": null,
      "max": null,
      "signed": false,
      "measurementMapping": {
        "series": "Motor",
        "sendMeasurementTemplate": 300,
        "type": "c8y_WorkingHours"
      },
      "number": "0x100",
      "divisor": 1,
      "multiplier": 5,
      "decimalPlaces": 2,
```

```
      "unit": "",
      "startBit": 32,
      "input": false,
      "name": "Working hours motor",
      "statusMapping": {
        "status": "read"
      },
      "noBits": 32,
      "offset": 0
    },
    {
      "min": null,
      "max": null,
      "signed": false,
      "measurementMapping": {
        "series": "Ventilator",
        "sendMeasurementTemplate": 300,
        "type": "c8y_WorkingHours"
      },
      "number": "0x101",
      "divisor": 1,
      "multiplier": 5,
      "decimalPlaces": 2,
      "unit": "",
      "startBit": 0,
      "input": false,
      "name": "Working hours ventilator",
      "statusMapping": {
        "status": "read"
      },
      "noBits": 32,
      "offset": 0
    },
    {
      "min": null,
      "max": null,
      "signed": false,
      "measurementMapping": {
        "series": "Ventilator",
        "sendMeasurementTemplate": 301,
        "type": "c8y_RevolutionMeasurement"
      },
      "number": "0x102",
      "divisor": 8,
      "multiplier": 1,
      "decimalPlaces": 0,
      "id": "25084050126511803",
      "unit": "rpm",
      "startBit": 0,
      "input": false,
      "name": "rpm ventilator",
      "statusMapping": {
        "status": "read"
      },
      "noBits": 16,
      "offset": 0
    },
    {
      "min": null,
      "max": null,
      "signed": false,
```

```
      "measurementMapping": {
        "series": "Motor",
        "sendMeasurementTemplate": 301,
        "type": "c8y_RevolutionMeasurement"
      },
      "number": "0x102",
      "divisor": 8,
      "multiplier": 1,
      "decimalPlaces": 0,
      "unit": "rpm",
      "startBit": 16,
      "input": false,
      "name": "rpm motor",
      "statusMapping": {
        "status": "read"
      },
      "noBits": 16,
      "offset": 0
    },
    {
      "min": null,
      "max": null,
      "signed": false,
      "measurementMapping": {
        "series": "Oil",
        "sendMeasurementTemplate": 302,
        "type": "c8y_TemperatureMeasurement"
      },
      "number": "0x103",
      "divisor": 1,
      "multiplier": 1,
      "decimalPlaces": 0,
      "unit": "C",
      "startBit": 0,
      "input": false,
      "name": "Temp oil",
      "statusMapping": {
        "status": "read"
      },
      "noBits": 8,
      "offset": -40
    },
    {
      "min": null,
      "max": null,
      "signed": false,
      "measurementMapping": {
        "series": "Motor",
        "sendMeasurementTemplate": 302,
        "type": "c8y_TemperatureMeasurement"
      },
      "number": "0x103",
      "divisor": 1,
      "multiplier": 1,
      "decimalPlaces": 0,
      "unit": "C",
      "startBit": 8,
      "input": false,
      "name": "Temp motor",
      "statusMapping": {
        "status": "read"
```

```
      },
      "noBits": 8,
      "offset": -40
    },
    {
      "min": null,
      "max": null,
      "signed": false,
      "measurementMapping": {
        "series": "Water",
        "sendMeasurementTemplate": 303,
        "type": "c8y_LevelMeasurement"
      },
      "number": "0x103",
      "divisor": 1,
      "multiplier": 1,
      "decimalPlaces": 0,
      "unit": "%",
      "startBit": 32,
      "input": false,
      "name": "Water level",
      "statusMapping": {
        "status": "read"
      },
      "noBits": 8,
      "offset": 0
    },
    {
      "min": null,
      "max": null,
      "eventMapping": {
        "text": "Oil level changed (0=low, 1=OK)",
        "eventTemplate": 304,
        "type": "c8y_OilLevelEvent"
      },
      "signed": false,
      "number": "0x103",
      "divisor": 1,
      "multiplier": 1,
      "decimalPlaces": 0,
      "unit": "",
      "startBit": 40,
      "input": false,
      "name": "Oil level",
      "statusMapping":{
      "status": "read"
      },
      "noBits": 8,
      "offset": 0,
      "enumValues": {
        "0": "Low",
        "1": "OK"
      }
    }
  ]
}
```

5.6.4. DEVICE MANAGED OBJECT

```
{
  "name": "J2ME_device_357666056160200 - Sweeper_v1",
  "type": "Sweeper_v1",
  "c8y_CanDevice": {
    "type": "/inventory/managedObjects/15266"
  }
}
```

Note that CAN devices do not need an address because identification of messages is purely based on CAN message id.

5.7. OPC-UA

OPC-UA is the interoperability standard for the secure and reliable exchange of data in the industrial automation space and in other industries. It is platform independent and ensures the seamless flow of information among devices from multiple vendors. The specification defined a standard set of objects, interfaces and methods for use in process control and manufacturing automation applications to facilitate interoperability. The most common OPC specification is OPC Data Access, which is used to read and write real-time data.
For details I refer you to OPC-UA specification. There is number of implementation of OPC-UA Servers provided by other companies like Kepware. For user manual please check **Cloud of Things' user guide**.

- In OPC-UA only registers are used (which on UI are named"Variables")
- There is no "startBit" and "noBit" configuration in device type, only "browsePath". property.
- There is no "parity", "stopBits", "dataBits", "baudRate" configuration on terminal side. Only "url" and credentials parameters.

In child device configuration there is only "name", "type" and "browsePath" configuration.

5.7.1. OPC-UA CONFIGURATION

For configuration of OPC-UA from user perspective it is worth to check **Cloud of Things' user guide**.

5.7.2. DEVICE TYPE MANAGED OBJECT

OPC UA type is stored as managed object and represents a certain model and version of device. IT is represented by a name which should be qualified enough to determine manufacturer, model and potentially version of the device. A device type has a set of properties which contains definition of referenced data. It is worth to mention that OPC UA specification doesn't recognize such type as device. So, choice of properties that are part of particular device depends only on interpretation of the user that defines device metadata type.

Example:

```
{
   "id":"14400",
   "name":"OPCUA_device_2.0",
   "type":"c8y_OPCUADeviceType",
   "c8y_Registers": [
      {
        "name": "simple_property_name",
        "category": "property_category",
        "browsePath": "path/to/simple/property",
        "attributeType": "Value",
        "statusMapping": { },
```

```
        "alarmMapping": {
            "raiseAlarmTemplate":300,
            "type":"c8y_DoorOpen",
            "text":"Door open",
            "severity":"MAJOR"
         },
         "measurementMapping":{
            "sendMeasurementTemplate":301,
            "type":"c8y_TemperatureMeasurement",
            "series":"T"
         },
         "eventMapping":{
            "eventTemplate":304,
            "type":"c8y_SampleBitEvent",
            "text":"Sample bit changed"
         }
      }, {
        "name": "simple_property_name",
        "category": "property_category",
        "browsePath": "path/to/complex/property",
        "dataType": "/inventory/managedObjects/14401"
      }, {
        "name": "event",
        "category": "property_category",
        "browsePath": "path/to/event_source",
        "attributeType": "EventNotifier",
        "eventType": "1",
        "eventFields": [
          "Message",
          "Severity"
        ],
        "alarmMapping": {
            "raiseAlarmTemplate":300,
            "type":"c8y_DoorOpen",
            "text":"Door open",
            "severity":"MAJOR"

      }
      }
   ]
}
```

Because OPC UA model is based on mesh network we need to be able to define hierarchical device model metadata which will be more complex and more generic than modbus device metadata. On the other side this metadata doesn't require any low-level configuration like "message id", "start bit" etc. In order to add new UI for OPC UA definition we need to add additional device type. Definition of this type will contain at least following fields:

- Device type: "OPC UA" in this case
- Unique name
- Set of variables:
    - Unique name
    - Display category: is used for grouping properties into sections in the visualization
    - Relative browse path to node, i.e. "4:Pipe/4:FlowTransmitter/4:Output" - needs to contain namespace index or namespace name
    - Node type, i.e. *Object*, *Variable*, *Method* (optional)
    - Attribute type (optional)
        - Simple attribute: attribute name (ie *Value*, *DisplayName, EventNotifier*) or
        - Complex attribute: other type name

- Multiplier and divisor: allows to handle arbitrary scaling
- Decimal places: allows shifting of values
- Unit: the unit of the measurement

5.7.3. TERMINAL MANAGED OBJECT

In Cloud of Things the OPC UA Gateway is represented as a managed object. It contains a OPC UA fragment. There is no need for additional configuration for the OPC UA Server on platform side.

Example:

```
{
  "name":"Gateway",
  "type":"opcua_gateway",
  "c8y_IsDevice":{ },
  "com_cumulocity_model_Agent":{ },
  "c8y_OPCUAGateway":{
    "url": "http://opcua.demo-this.com:51211/UA/SampleServer",
    "applicationUri": "urn:localhost:UA:TestClient",
    "productUri": "urn:cumulocity.com:UA:c8y_OPCUAGateway"
    "transmitRate": 12,
    "pollingRate": 12,
    "userIdentityName": "login",
    "userIdentityPassword": "password"
  }
}
```

- url: opc server url
- applicationUri: application identity, default is urn:localhost:UA:c8y_OPCUAGateway (configurable via properites file)
- productUri: product identity, default is "urn:cumulocity.com:UA:c8y_OPCUAGateway" (configurable via properties file)
- transmitRate: how often data is send from gateway to platform, by default will be send immediately
- pollingRate: how often data is send from opcua server to gateway, by default the used prosys library will decide
- userIdentity: credentials

5.7.4. DEVICE MANAGED OBJECT

Any node on OPC UA Server side might be interpreted as connected device. The connected devices are represented as child managed objects of the gateway managed object. So, they have to contain at least the browse path to corresponding nodes.

Example:

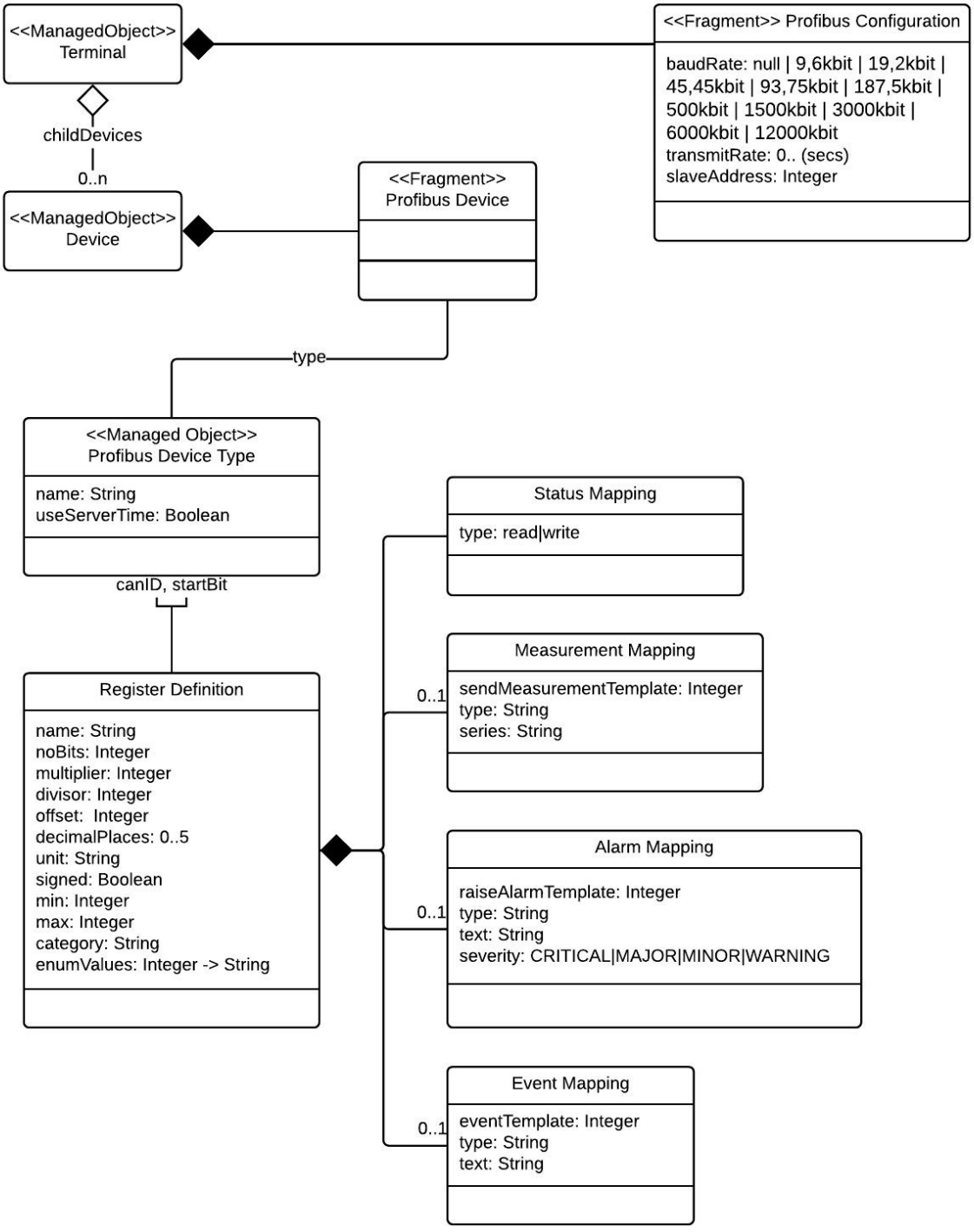
```
{
  "name":"Pump",
  "type":"c8y_OPCUADevice",
  "c8y_OPCUADevice":{
    "browsePath": "4:Boilers/4:Boiler #1",
    "type":"/inventory/managedObjects/14400"
  }
}
```

- Name
- Absolute *BrowsePath* to node, ie “4:Boilers/4:Boiler #1” - needs to include namespace index or namespace name
- Device type

5.8. PROFIBUS DP

The Profibus DP protocol integration does not provide functionality for coils, discrete inputs and input register. Since Profibus DP does not differentiate between these kinds of data sources the concept of logical holding registers is simply applied to Profibus registers. Similar to the CAN integration the Profibus terminal will simply listen to messages on the bus. The messages must be sent to the terminal's configured slave address by a component on the Profibus DP network. This will typically require programming on the respective PLC.

5.8.1. PROFIBUS SPECIFIC DOMAIN MODEL



5.8.2. TERMINAL MANAGED OBJECT

```
{
 "name":"SMARTbox Terminal",
 "type":"SMARTbox",
 "com_cumulocity_model_Agent":{},
 "c8y_IsDevice":{},
 "c8y_ProfibusConfiguration":{
   "baudRate":1500000,
   "slaveAddress":24,
   "transmitRate":5
 }
}
```

5.8.3. DEVICE MANAGED OBJECT

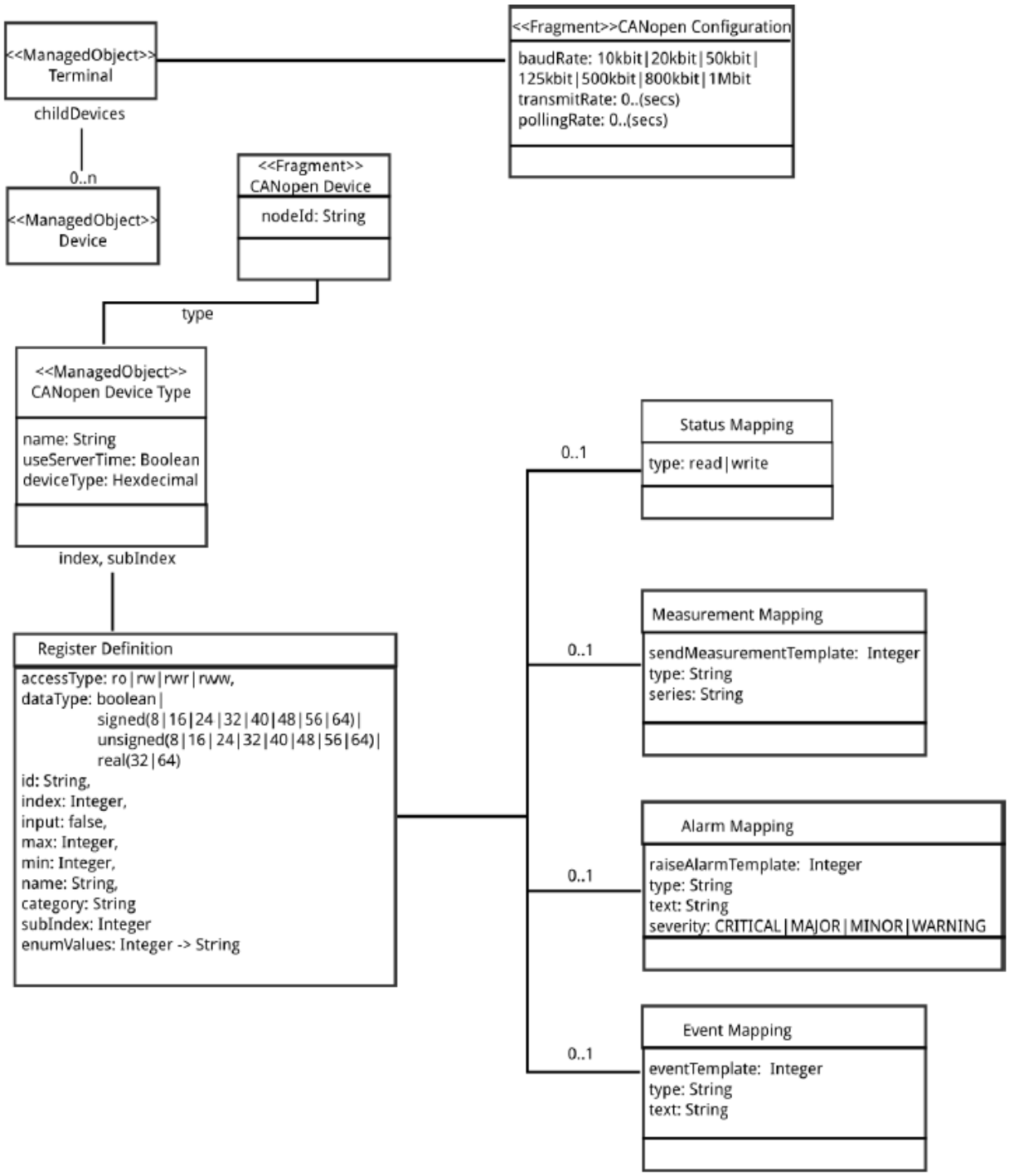
```
{
 "name":"DP",
 "type":"SPI_Profibus2",
 "c8y_ProfibusDevice":{
 "type":"/inventory/managedObjects/3527246"
  }
 "c8y_FieldbusChild":{},
}
```

5.9. CANOPEN

CANopen is a CAN-based communication system. It comprises higher-layer protocols and profile specifications. CANopen has been developed as a standardized embedded network with highly flexible configuration capabilities. In the CANopen data model, devices must have an Object Dictionary, which is used for configuration and communication with the device. An entry in the object dictionary is defined by a 16-bit index and 8-bit sub-index. The basic data types included in the object dictionary are: Boolean, unsigned integer, signed integer and floating point.

- In CANopen, only registers are used (named “Variables” on UI), which is defined by the 16-bit index and 8 bit sub-index, to represent an entry in the Object Dictionary.
- There is no “startBit”, “noBits”, “multiplier”, “divisor”, “decimalPlaces”, “input”, “signed” properties in register definition. In contrast, it has “index”, “subIndex”, “dataType”,“accessType” properties.
- There is no “parity”, “stopBits”, “dataBits” configuration on terminal side, only “baudRate” is applicable.
- In adding child devices, there is only “name”, “type” and “nodeId” properties.

5.9.1. CANOPEN SPECIFIC DOMAIN MODEL



5.9.2. DEVICE TYPE MANAGED OBJECT

The CANopen Object Dictionary is stored as an device type in the device database in Cloud of Things. The properties and their description in a register definition is explained in the following table:

Name	Description
name	Human readable name for visualization
category	Used for grouping variable into sections in the vizalization
index	Index of the variable in the OD of the device (16-bit)
subIndex	Sub-Index of the variable in the OD of the device (8-bit)
datatype	● boolean ● unsigned[8/16/24/32/40/48/56/64] ● signed[8/16/24/32/40/48/56/64] ● real32, real64
accessType	● read-only (ro) ● write-only (wo) ● read-write (rw) ● read-write on process input (rwr) ● read-write on process output (rww) ● const
unit	Logical unit of the variable
min	Minimum value
max	Maximum value

Example:


```
{
"id": "502" ,
"name": "First" ,
"type": "c8y_CANopenDeviceType"
"deviceType": 3800099 ,
"fieldbusType": "canopen" ,
"fieldbusVersion": 4 ,
"c8y_useServerTime": false ,
"c8y_IsDeviceType": {},
"c8y_Registers": [
{
"accessType": "rw" ,
"dataType": "real32" ,
"id": "8009630535187862" ,
"index": 8193 ,
"input": false ,
"max": 100 ,
"measurementMapping": {
"sendMeasurementTemplate": 301 ,
"series": "T" ,
"type": "Temp"
},
"min": -100 ,
"name": "p2001_0" ,
"statusMapping": {
"status": "write"
},
"subIndex": 0
},
{
"accessType": "rw" ,
"alarmMapping": {
"mask": 3 ,
```

```
"raiseAlarmTemplate": 300 ,
"severity": "MAJOR" ,
"text": "Temp Alarm Text" ,
"type": "Temp"
},
"dataType": "unsigned8" ,
"id": "7985277938590234",
"index": 8194 ,
"input": false ,
"max": 255 ,
"min": 0 ,
"name": "p2002_0" ,
"statusMapping": {
"status": "write"
},
"subIndex": 0
},
{
"accessType": "ro" ,
"dataType": "signed16" ,
"eventMapping": {
"eventTemplate": 302 ,
"text": "Temp1 Event" ,
"type": "Temp1"
},
"id": "3956317985243738" ,
"index": 24576 ,
"input": false ,
"name": "p6000_1" ,
"statusMapping": {
"status": "read"
},
"subIndex": 1
},
{
"accessType": "rw" ,
"dataType": "signed16" ,
"eventMapping": {
"eventTemplate": 303 ,
"text": "Temp2 Event" ,
"type": "Temp2"
},
"id": "25937248862216955" ,
"index": 24576 ,
"input": false ,
"max": 32767 ,
"min": -32768 ,
"name": "p6000_2" ,
"statusMapping": {
"status": "write"
},
"subIndex": 2
},
{
"accessType": "rw" ,
"dataType": "unsigned64" ,
"id": "19457998089956807" ,
"index": 24577 ,
"input": false ,
"max": 4295032831 ,
"min": 0 ,
```

```
"name": "p6001_0" ,
"statusMapping": {
"status": "write"
},
"subIndex": 0
}
]
}
```

5.9.3. TERMINAL MANAGED OBJECT

The terminal normally represents a gateway which runs the CANopen agent. The agent needs to manifest to Cloud of Things that it has CANopen support with the following fragment:

```
{
"c8y_CANopenConfiguration":
{
"baudRate": 800 ,
"pollingRate": 3 ,
"transmitRate": 5
}
}
```

And also adding following capabilities to the c8y_SupportedOperations:

```
{
"c8y_SupportedOperations":
[
"c8y_CANopenAddDevice" ,
"c8y_CANopenRemoveDevice" ,
"c8y_CANopenConfiguration"
]
}
```

5.9.4. DEVICE MANAGED OBJECT

Any CANopen node on the CAN network that you want to communicate with is interpreted as a child device of the terminal in Cumulocity. Required fragments for a CANopen child device is as following:

```
{
"name": "First" ,
"id": "110296" ,
"c8y_CANopenDevice":
{
"nodeId": 5 ,
"type": "/inventory/managedObjects/502"
},
"c8y_SupportedOperations": [
"c8y_Command"
]
}
```

Note that in addition to the normal “c8y_CANopenDevice” fragment, also “c8y_Command” must be added into “c8y_SupportedOperations” since CANopen devices are required to support the device shell feature.

6. THE SMARTREST PROTOCOL

SmartREST provides the base for secure and wide-area capable fieldbus communications. It provides:

- A scalable transport protocol between terminal and remote management service that works
  - in any IP-based network topology, fixed and mobile.
- Confidentiality through encryption and protection against man-in-the-middle attacks.
- Authentication and authorization of devices.
- Secure and reliable communication towards fieldbus devices over wide-area networks, regular as well as real-time.
- Availability management.
- Plug-and-play installation of new terminals.
- Protocol modularity and extensibility.

SmartREST is based on HTTP(S) POST requests. The structure of each request is

```
POST /s HTTP/1.0
Authorization: Basic <password>
X-ID: <name of SmartREST template collection>
Content-Length: <length>

<templateNumber>,<msgData>,<msgData>, …
<tem-
plateNumber>,<msgData>,<msgData>
, …  …
```

The parameters in the request are:

- password: base64-encoded credentials. Initial credentials for the device can be automatically obtained through the Cloud of Things Device Credentials API.
- X-ID: contains the name of the SmartREST template collection
- templateNumber: Identifies the SmartREST template to be applied (see below).
- msgData: Data according to the template in use.

The structure of a response is:

```
HTTP/1.1 200 OK
Content-Length: <length>
<templateNumber>,<msgData>,<msgData>, …
<templateNumber>,<msgData>,<msgData>, …
...
```

In the response, a template number is provided in as similar way as in a request:

- templateNumber: Identifies the SmartREST template that was used to build the response.
- line: The request line that this response corresponds to.
- msgData: Data according to the template in use.

7. SMARTREST TEMPLATES FOR FIELDBUS COMMUNICATION

For Modbus and Canbus Fieldbus device type, a set of SmartREST templates are automatically set up to simplify the communication between the terminal and Cloud of Things. The following sections describe the communication protocol for operating Modus devices.
For OPC-UA devices SmartREST is not supported.

7.1. SEND COIL STATUS

There is one fixed template per Fieldbus device type for sending the status of all readable coils to Cloud of Things. This template is generated with the template number 100 and assumes the following request body:
`100,<deviceId>,<coilValue>,<coilValue>, …`

The expected ordering of the values is:

1. All Coils in ascending order by their number.
2. All Discrete inputs ascending order by their number.

In the above example, there are only two coils "Operating mode" and "Power" with Numbers 1 and 2. assume that the operating mode for device 12345 is set to manual and the power is turned on. To send a status update, use the following SmartREST request:

```
POST /s HTTP/1.0
Authorization: Basic <password>
X-ID: WiloVarioCOR_1.0
Content-Length: 15

100,12345,0,1
```

7.2. SEND REGISTER STATUS

Similarly, there is one fixed template per Fieldbus device type for sending the status of all logical registers to Cloud of Things. This template is generated with the template number 101 and assumes the following request body:

```
101,<deviceId>,<registerValue>,<registerValue>, …
```

The expected ordering of the values is:

1. All holding registers in ascending order by their number, with their starting bit in ascending order as secondary criteria.
2. All input registers in ascending order by their number, with their starting bit in ascending order as secondary criteria.

E.g., assume that in the above example, the holding register with the number 2 contains the value 10. The register definitions specify that the first three bits starting from bit 0 represent a logical register, and the fourth bit represents a logical register as well. Hence, the corresponding status update would be (only request body shown):

```
101,12345,2,1
```

(Since the first three bits of the value 10 are 2, and the fourth bit is 1.)

7.3. UPDATE COIL

The terminal receives operations to update a coil using a template with the fixed number 200 and the following format:

```
200,<operationId>,<ipAddress>,<deviceAddress>,<input>,<coil-
Number>,<coilValue>
```

Assuming the above example, the terminal would receive the following operation when the user would set the "operating mode" to "automatic":

```
200,12345,,1,false,1,1
```

The first "1" corresponds to the Modbus device address (c8y_ModbusDevice.address). The boolean flag “false” signals that it is a coil (and not a discrete input). The second "1" corresponds to the number of the "operating mode" coil (c8y_Coils[0].number). Finally, the third "1" corresponds to setting the mode to "automatic" (= true). The example shows a serial Modbus device, hence the IP address field is shown as empty.

7.4. UPDATE REGISTER

Correspondingly, the operation to update a register is using a template with the fixed number 201 and this format:

```
201,<operationId>,<ipAddress>,<deviceAddress>,<input>,<register-
Number>,<startBit>,<noBits >,<registerValue>
```

Assuming the above example, the terminal would receive the following operation when the user would set the "Sample enum" to "Mode4":

```
201,12345,,1,false,2,0,3,4
```

Note that when the terminal receives a request to update a logical register on the Fieldbus device that does not correspond to a full physical register (noBits < length), it first has to read the physical register, then apply the logical value to the relevant bits and write it back.

7.5. RAISE ALARM

To raise an alarm, use the template referenced in the "raiseAlarmTemplate" property of the corresponding coil or register. The general format is:

```
<raiseAlarmTemplate>,<deviceId>[,time]
```

The time is required in case the device supports buffering of readings and may send an alarm later than it actually occurred. In the above example, a "Door open" alarm would be sent as follows:

```
300,12345,2014-10-31T12:03:27.745Z
```

“time” can be omitted if “useServerTime” is checked.

7.6. LOG EVENTS

Events are logged using the template referenced by the “eventTemplate” property of the corresponding coil or register. The format is:

```
<eventTemplate>,<deviceId>[,time],<coil/registerValue>
```

“time” can be omitted if “useServerTime” is checked.

7.7. SEND MEASUREMENT

To send a measurement, use the template referenced in the "sendMeasurementTemplate" property of the corresponding register. The format is:

```
<sendMeasurementTemplate>,<deviceId>,<time>,<registerValue>,...
```

In the above example, a register value of 2350 of a temperature sensor would be sent like this:

```
301,12345,2014-10-31T12:04:21.1245Z,23.5
```

Note that the temperature definition defined two decimal places, which makes 2350 correspond to 23.5 degrees C. Note also that if multiple registers are using the same type (measurementMapping.type), they need to be sent in one request. If “useServerTime” is enabled, the time stamp can be omitted.

8. USING SMARTREST FROM THE TERMINAL

Now that the basic protocol mechanisms have been defined, this section discusses the process of setting up and operating devices.

8.1. ADDING A DEVICE

When a new device is connected, the terminal receives an operation as follows: `202,<line>,<operationId>,<deviceId>,<deviceAddress>,<fieldbusDeviceType>`

with these parameters:

- operationId: The ID of the operation that was sent to the terminal. This ID is used for updating the operation in Cloud of Things to indicate that the operation is now executing or has been successfully executed.

- deviceId: The ID of the new device that should be used when sending the device's measurements, alarms, etc. to Cloud of Things. It will also be used by Cloud of Things when the user wants to send an operation to the device.
- The Fieldbus address of the device. On Modbus for example this is the Modbus address of the device
- The Fieldbus device type. This is sent in the form of a URI to the Device Database entry.

As an example, the following operation is sent when adding the pump:
`202,1,45678,34567,1,/inventory/managedObjects/23456`

Whenever the terminal picks up the operation and executes it, it marks the operation in Cloud of Things as "EXECUTING" by sending the following template:
`102,<operationId>`

The terminal now queries the Device Database using the URI. Using the above data and the information from the Device Database, the terminal is now ready to fully operate the device. If the process succeeds, the terminal marks the operation as "SUCCESSFUL" by sending:
`103,<operationId>`

This process is used for all operations sent to the terminal

8.1.1. CREATING NEW FIELDBUS CHILD DEVICES

Fieldbus Application creates the respective child device in inventory. The managed object ID of the newly created child is delivered to the terminal via the deviceId parameter in the operation. The terminal just has to use this child ID when sending data for this particular device.

8.1.2. ADDING MODBUS/TCP CHILD DEVICE

In case of Modbus/TCP child devices there is additional c8y_ModbusDevice.ipAddress field in this operation.

8.1.3. ADDING CANOPEN CHILD DEVICE

In case of CANopen child devices, the <deviceAddress> field is interpreted as the node ID of a CANopen node.

8.2. OPERATING A DEVICE

To operate a device, the terminal carries out change detection, regular measurement sending and execution of operations.
For change detection, the terminal executes the following loop using the polling rate defined for the terminal:

- For each connected device
  - Poll all coils and (physical) registers.
  - Compare the current polling results with the previous polling results.

  In case of a difference,
    - Send a status update for the complete set of coils or registers to Cloud of Things.
    - Raise alarms where an alarm mapping is defined.
    - Send events where an event mapping is defined.

For measurement sending, the terminal polls all registers with "sendMeasurementTemplate" with an interval defined in the "transmitRate" property of the terminal.
For execution of operations, the terminal listens to notifications from Cloud of Things ("long polling"). Whenever operations are sent by the user, the terminal executes the operations and then continues to listen to further operations. After the operation is executed, the terminal sends the operation result to Cloud of Things. It also updates the inventory with the current state of the coils (when a coil was updated) or registers (when a register was updated).

8.2.1. REMOVING A DEVICE

When the device is disconnected or removed, a remove operation is send to inform the terminal about the child device removal. **Note** the remove operation currently only applies to CANopen protocol.

8.2.2. REMOVING A CANOPEN CHILD DEVICE

The remove operation for CANopen is defined as following:

```
204,<line>,<operationId>,<deviceId>,<nodeId>
```

9. VISUALIZATION

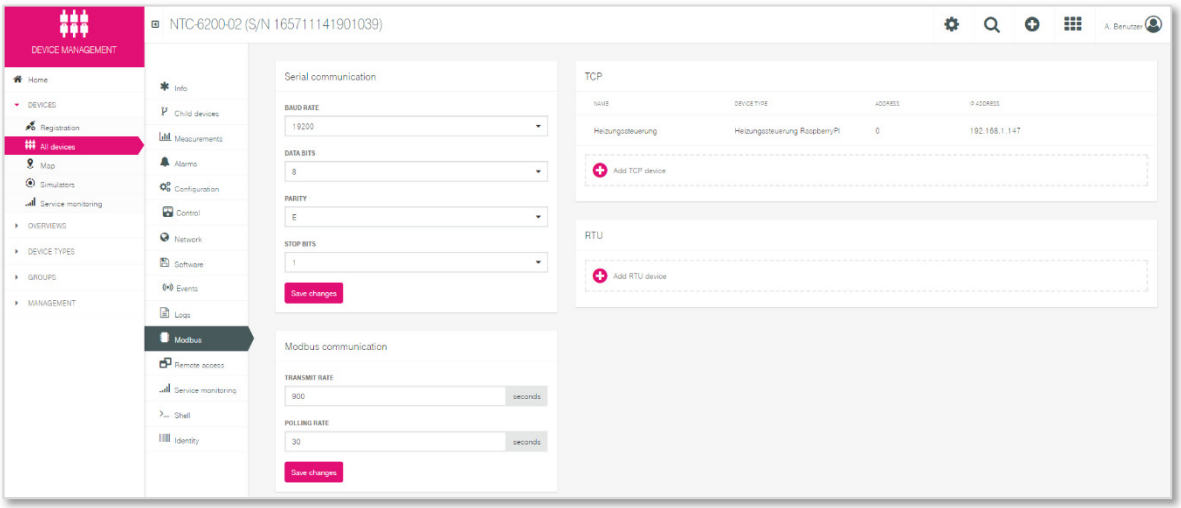
The user interface supports three main workflows:

- Technician workflow: Configure Fieldbus setup in Cloud of Things, after preparing the actual physical devices and connecting them to the terminal.
- Device implementation workflow: Configure Fieldbus parameters of a device type in Cloud of Things's Device Database.
- Operation workflow: Operating Fieldbus device using normal Cloud of Things means plus additional status/configuration-related widgets.

There is an alternative workflow for expert technicians who would like to set up the devices but also need to customize the Device Database so that only a subset of the parameters is actually handled by the terminal.

9.1. TECHNICIAN WORKFLOW

After the all devices have been connected and the terminal is registered with Cloud of Things, the technician navigates to the terminal in Cloud of Things and selects the tab for the required protocol. At the moment there are separate tabs Modbus and for CAN that will be displayed based on the communicated capabilities of the device. On the respective tab, the technician can configure the serial communication parameters (if applicable) and the Fieldbus communication parameters.



The technician can also add Fieldbus devices to the configuration. These Fieldbus devices are created as child managed objects in the inventory and linked to the Device Database using a parameter in their protocol specific Fieldbus device fragment. The default name of the child devices can be constructed from the name of the Fieldbus device type and address.

To support the alternative workflow, the device type could be displayed as a link to the corresponding page in the Device Database editor (see below).

9.2. DEVICE IMPLEMENTATION WORKFLOW

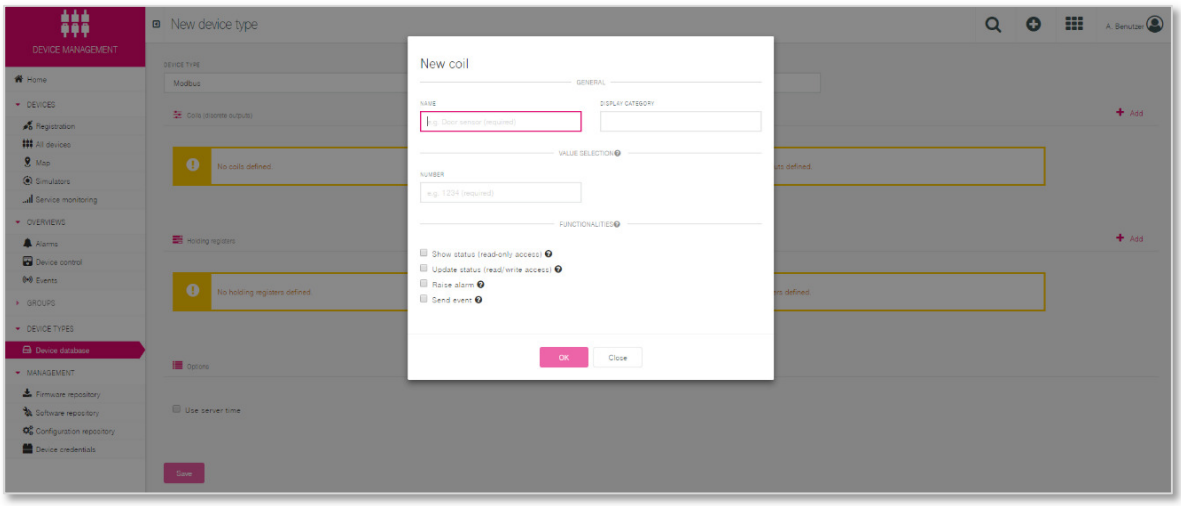
For setting up device types, a "Device Database" is provided in the administration tool as sketched below. This tool allows configuration of Fieldbus device types. When editing a device type, coils and registers can be specified for the device type.

The properties of coils and registers and their handling needs to be specified. These properties have been previously discussed already.
Note that the user only has to specify the functionality that is desired to have for a coil or register (such as raising an alarm or similar). The user interface generates behind the scenes a set of SmartREST template for the device type, just like other devices do.

9.2.1.1. ADDING A COIL DEFINITION

Click **Add** at the top right of the **Coils (discrete inputs)** section, to add a coil definition. This will open a dialog to specify the coil. Enter the following information:

1. Enter the name of the coil as being displayed in the user interface.
2. Optionally, enter the display category to structure your data in widgets.
3. Enter the number of the coil in the Modbus device.
4. Select the **Show status** checkbox if you want to show the coil's current value in the Fieldbus Device Widget. In this case, you can enter the text that the Fieldbus Device Widget should show for unset and set coils.
5. Select the **Update status** checkbox if you want to be able to edit the coil from the Fieldbus Device Widget.
6. Select the **Raise alarm** checkbox if an alarm should be raised when the coil is set in the device. In this case, you can specify the type of the alarm that is raised, its text and its severity. Note that there can only be one alarm active of a particular type for a particular device.
7. Select the **Send event** checkbox if an event should be generated each time the value of the coil changes. If **Send event** is selected, you can specify the type of event and the text in the event.
8. Click **OK** to finish editing the coil.

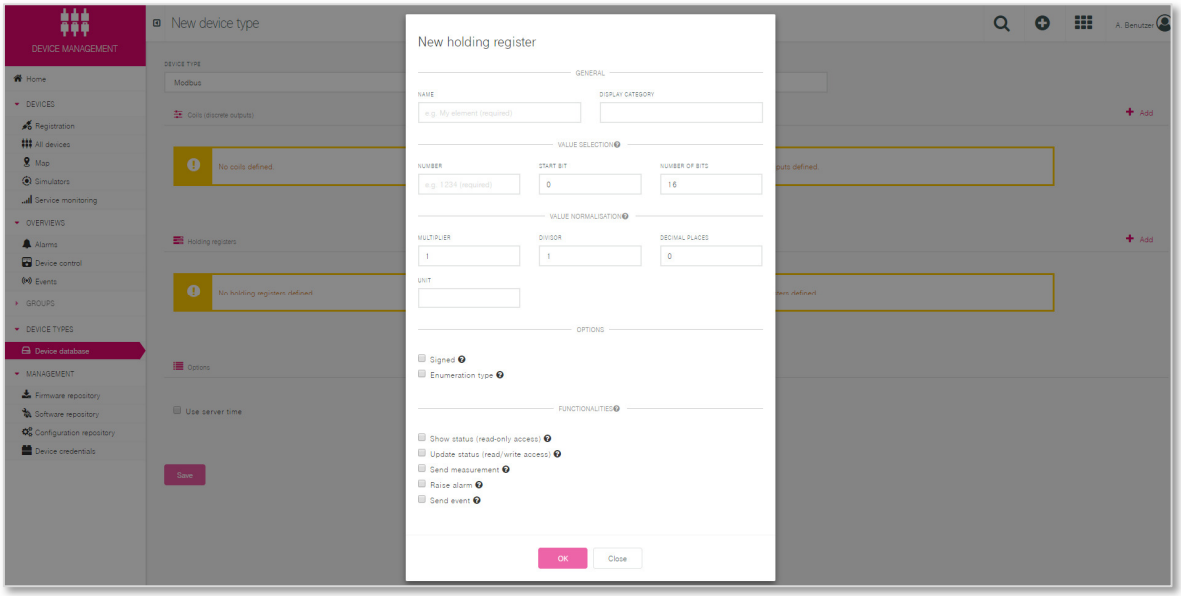


The same functions are available for discrete inputs. However, it is not possible to update the status of a discrete input.

9.2.1.2. ADDING A REGISTER DEFINITION

Click **Add** at the top right of the **Holding registers** section, to add a register definition. This opens a dialog to enter the details of the register definition:

1. Enter the name of the register being displayed in the user interface.
2. Optionally, enter the display category to structure your data in widgets.
3. Enter the number of the register in the Modbus device. You can indicate a subset of bits to be used from a register by providing a start bit and a number of bits. This allows you to split a physical Modbus register into a set of "logical registers".
4. To scale the integer value read from the Modbus device, you can enter a multiplier, a divisor and a number of decimal places. The register value is first multiplied by the "multiplier", then divided by the "divisor" and then shifted by the number of decimal places. Note, that the terminal may use integer arithmetic to calculate values sent to Cloud of Things. For example, if you use a divisor of one and one decimal place, a value of 231 read from the terminal will be sent as 23.1 to Cloud of Things. If you use a divisor of ten and no decimal places, the terminal may send 23 to Cloud of Things (depending on its implementation).
5. Indicate the unit of the data, for example, "C" for temperature values.
6. Select the **Signed** checkbox if the register value should be interpreted as signed number.
7. Select the **Enumeration type** checkbox if the register value should be interpreted as enumeration of discrete values. If **Enumeration type** is selected, you can click **Add value** to add mappings from a discrete value to a text to be shown for this value in the widget. Click **Remove value** to remove the mapping.
8. Select the **Show status** checkbox if you want to show the current value of the register in the Fieldbus Device Widget.
9. Select the **Update status** checkbox if you want to be able to edit the register from the Fieldbus Device Widget. If **Update status** is selected, two additional fields **Minimum** and **Maximum** appear. Using these fields, you can constrain numerical values entered in the widget.
10. Select the **Send measurement** checkbox if you want the values of the register to be regularly collected according to the transmit interval. In this case, add a measurement type and a series to be used. For each measurement type, a chart is created in the **Measurements** tab. For each series, a graph is created in the chart. The unit is used for labelling the measurement in the chart and in the Fieldbus Device Widget.
11. Select the **Raise alarm** checkbox if an alarm should be raised when the register is not zero in the device measurement. In this case, you can specify the type of the alarm raised, its text and its severity. Note, that there can only be one alarm active of a particular type for a particular device.
12. Select the **Send event** checkbox if an event should be generated each time the value of the register changes. If **Send event** is selected, you can specify the type of event and the text in the event.
13. Click **OK** to save your settings.



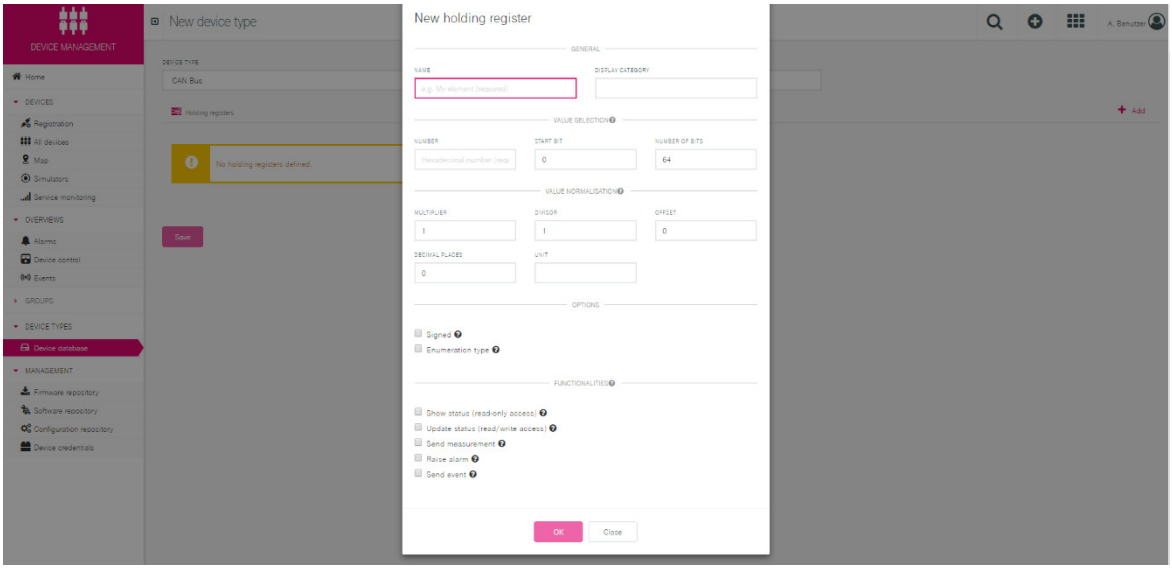
In the **Options** section, select the checkbox **Use server time** to create the time stamps for data on the server instead of on the terminal. If you need to support buffering of data on the terminal, leave this checkbox clear.
Finally, click **Save** to save your settings.
If you edit a device type that is currently in use, you may need to

- restart the terminals that use the device type,
- reconfigure dashboards and widgets that use the device type

9.2.2. CONFIGURING CAN BUS DATA

CAN device types can be configured in a very similar way as Modbus device types. For more information, see **Configuring Modbus data** above. The differences are:

- Holding registers are used to describe the different pieces of data inside CAN messages.
- Enter the CAN message ID of the specific message the data should be extracted from. Use a hexadecimal number for the message ID.
- Conversion of values is extended by an offset parameter. This will be added or substracted from the register value, depending on its sign. The offset calculation is done after applying multiplier and divisor, and before performing decimal shifting.

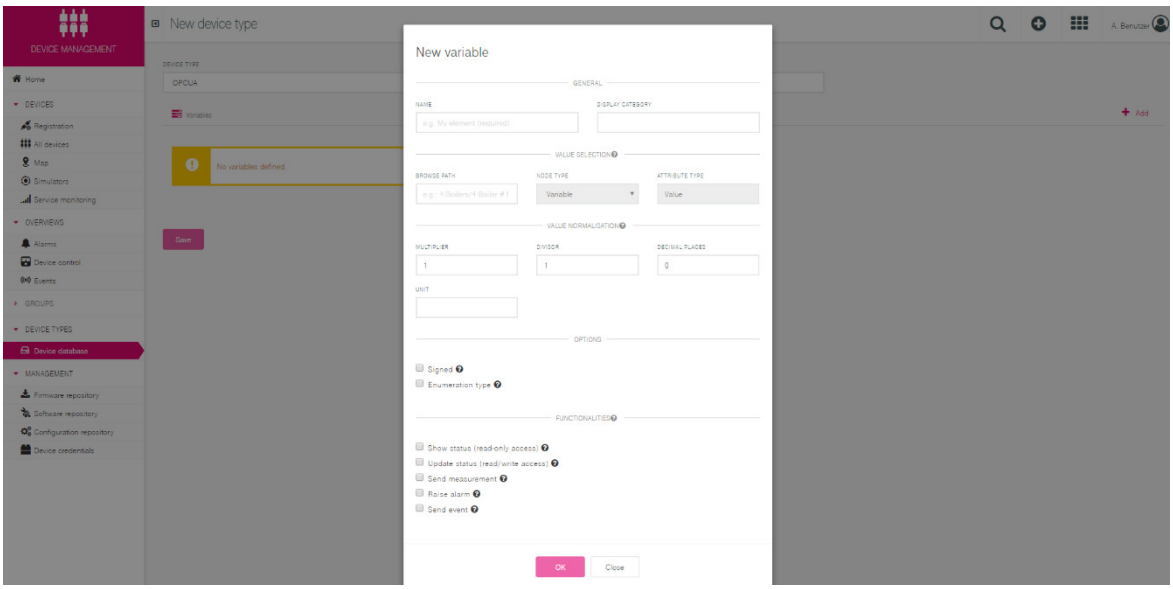


9.2.3. CONFIGURING OPC UA

OPC UA device types can be configured in a very similar way as Modbus device types. For more information, see **Configuring Modbus data** above.
The main difference is how data is addressed. OPC UA servers provide a hierarchical object model of connected nodes. The nodes are addressed by the browse path from the root of the object model to the respective node.
To simplify configuration, the browse path is split into two parts in Cloud Fieldbus:

- From the root to the OPC UA device.
- From the OPC UA device to a node with data of that device.

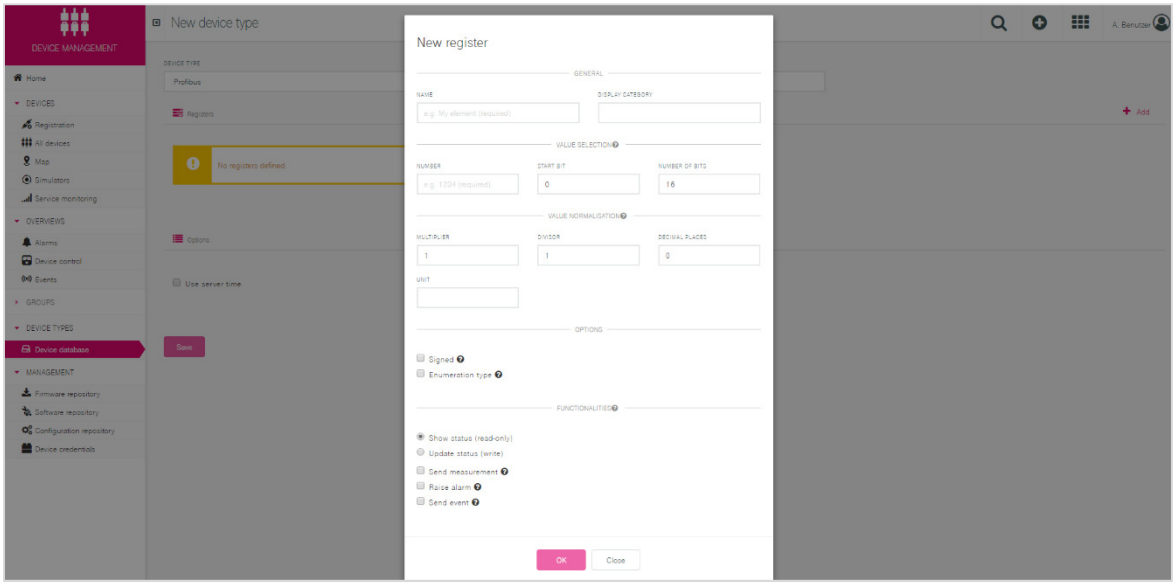
When you click **Add**, enter the second part of the path into the **field as shown in the image below. Note that the OPC UA agent currently only supports nodes of type "Variable". The description of the paths should be either provided with your OPC UA server or with your devices.



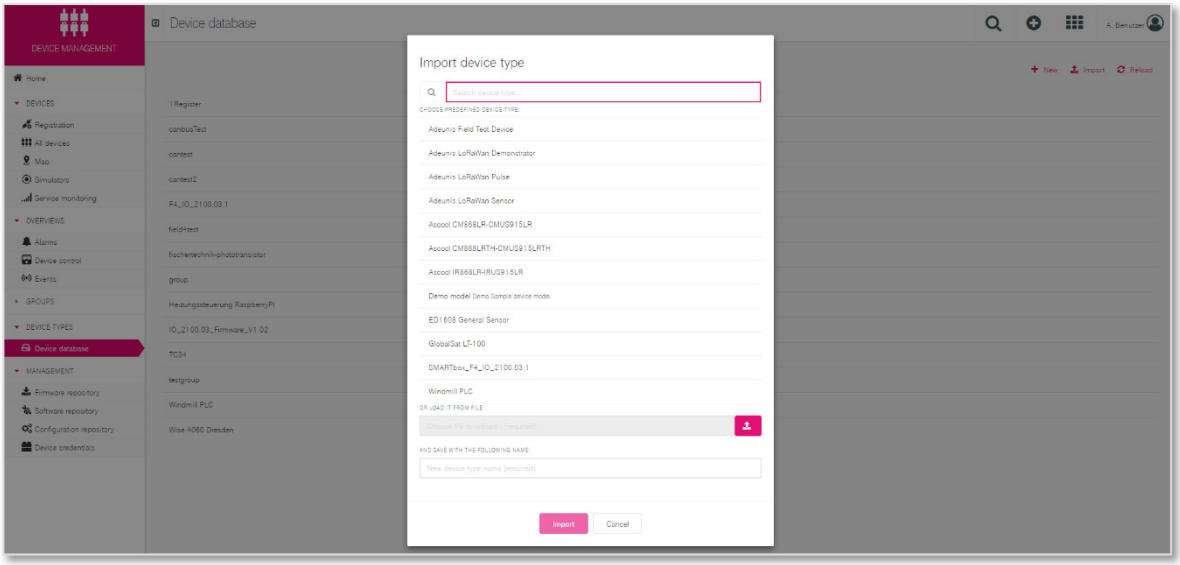
9.2.4. CONFIGURING PROFIBUS DATA

To configure a Profibus device type, select "Profibus" as device type from the dropdown list and enter a name for it.
In the Register section, click **Add** at the right to add one or more register definitions as described exemplarily for Modbus devices in **Adding a register definition** above.
In the **Options** section, select the checkbox **Use server time** to create the time stamps for data on the server instead of on the terminal. If you need to support buffering of data on the terminal, leave this checkbox clear.
Finally, click **Save** to save your settings.
If you edit a device type that is currently in use, you may need to

- restart the terminals that use the device type,
- reconfigure dashboards and widgets that use the device type.

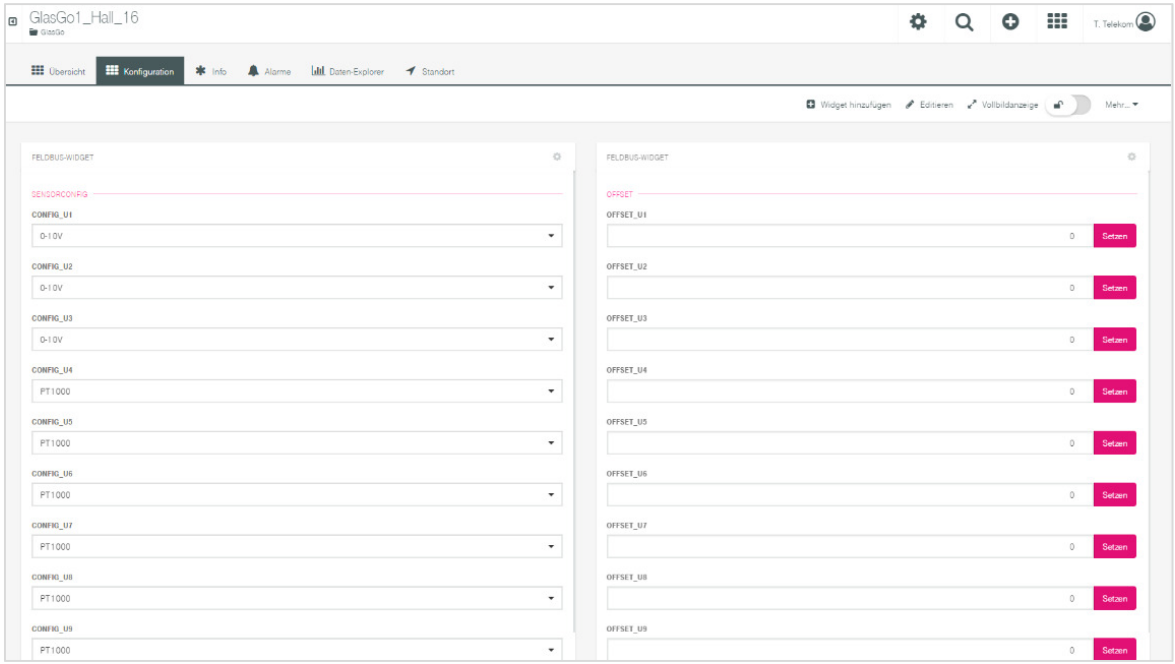


Some hardware suppliers provide their own json file for the fieldbus configuration. In this case the user only has to import the file via the import functionality in the device database.



9.3. OPERATION WORKFLOW

A Dashboard widget (Fieldbus device widget) to show all coils and registers with sendStatusTemplate, using either the definitions from enumValues or the literal value is provided. Coils and registers are editable, if updateStatusTemplate is set. In this case, the corresponding command is generated.
The device then receives the command, executes the configuration change and sends the result of the operation (successful/failed) back to Cloud of Things. Only when the configuration change has been executed successfully by the device, the coil or register values are displayed in Cloud of Things according to this configuration.



10. REFERENCES

10.1. SEND COIL STATUS (GENERATED FROM METADATA)

10.1.1. SENSOR LIBRARY STRUCTURE

```
{
  "id": <deviceId>,
  "c8y_CoilStatus": {
    "<property>": 0|1,
    "<property>": 0|1,
    ...
  }
}
```

10.1.2. SMARTREST

```
100,<deviceId>,<coilValue>,<coilValue>, …
```

10.1.3. SMARTREST REQUEST TEMPLATE

```
{
  "msgId":"100",
  "method":"PUT",
  "contentType":"application/vnd.com.nsn.cumulocity.managedObject+json",
  "placeholder":"%",
  "resourceUri":"/inventory/managedObjects/%",
  "templateString":
"{\"c8y_CoilStatus\":{\"<property>\":\"%\",\"<property>\":\"%\", ... }}",
  "values": [ "UNSIGNED", "UNSIGNED", "UNSIGNED", ... ]
}
```

10.2. UPDATE COIL

10.2.1. OPERATION

```
{
  "id": <operationId>,
  "c8y_SetCoil":{
    "ipAddress": <ip address or empty>,
    "address": <Modbus address>,
    "coil": <coil number>,
    "value": <0|1>
  }
}
```

10.2.2. SMARTREST

```
200,<operationId>,<ipAddress>,<modbusAddress>,<coilNumber>,<0|1>
```

10.2.3. SMARTREST RESPONSE TEMPLATE

```
{
  "condition":"$.c8y_SetCoil",
  "msgId":"200",
  "pattern":[
    "$.id",
    "$.c8y_SetCoil.ipAddress",
    "$.c8y_SetCoil.address",
    "$.c8y_SetCoil.coil",      "$.c8y_Set-
Coil.value"
  ]
}
```

10.3. SEND REGISTER STATUS (GENERATED FROM METADATA)

10.3.1. SENSOR LIBRARY STRUCTURE

```
{
  "id": <deviceId>,
  "c8y_RegisterStatus": {
    "<property>": <value>,
    "<property>": <value>,
    ...
  }
}
```

10.3.2. SMARTREST

```
101,<deviceId>,<value>,<value>, …
```

10.3.3. SMARTREST REQUEST TEMPLATE

```
{
  "msgId":"101",
  "method":"PUT",
  "contentType":"application/vnd.com.nsn.cumulocity.managedObject+json",
  "placeholder":"%",
  "resourceUri":"/inventory/managedObjects/%",
  "templateString":
"{\"c8y_RegisterStatus\":{\"<property>\":\"%\",\"<property>\":\"%\",  ...
}}",
"values": [ "UNSIGNED", "SIGNED", "SIGNED", ... ]
}
```

10.4. UPDATE REGISTER (FIXED)

10.4.1. SENSOR LIBRARY

```
{
  "id": <operationId>,
  "c8y_SetRegister":{
    "ipAddress": <ip address or empty>,
    "address": <Fieldbus address>,
    "register": <register number>,
    "startBit": <start bit>,
    "noBits": <number of bits>,
    "value": <register value>
  }
}
```

10.4.2. SMARTREST

```
201,<line>,<operationId>,<ipAddress>,<modbusAddress>,<register-
Number>,<startBit>,<noBits>,<registerValue>
```

SMARTREST RESPONSE TEMPLATE

```
{
  "condition":"$.c8y_SetRegister",
  "msgId":"201",
  "pattern":[
    "$.id",
    "$.c8y_SetRegister.ipAddress",
    "$.c8y_SetRegister.address",
    "$.c8y_SetRegister.register",
    "$.c8y_SetRegister.startBit",
```

```
    "$.c8y_SetRegister.noBits",
    "$.c8y_SetRegister.value"
  ]
}
```

10.5. RAISE ALARM (GENERATED)

10.5.1. SMARTREST

```
<raiseAlarmTemplate>,<deviceId>,<time>
```

10.5.2. SMARTREST REQUEST TEMPLATE

```
{
  "msgId":"<raiseAlarmTemplate>",
  "method":"POST",
  "contentType":"application/vnd.com.nsn.cumulocity.alarm+json",
  "placeholder":"%",
  "resourceUri":"/alarm/alarms",
  "templateString":
"{\"source\":{\"id\":\"%\"},\"type\":\"<type>\",\"status\":\"AC-
TIVE\",\"sever                                          ity\":\"<sever-
ity>\",\"time\":\"%\",\"text\":\"<text>\"}",
  "values": [ "UNSIGNED", "DATE" ]
}
```

Note: <time> can take either date value from device (if c8y_useServerTime is false) or NOW otherwise (then server time will be used on Cloud of Things platform).

10.6. SEND EVENT (GENERATED)

10.6.1. SMARTREST

```
<eventTemplate>,<deviceId>,<time>,<coil/registerVale>
```

10.6.2. SMARTREST REQUEST TEMPLATE

```
{
  "msgId":"<eventTemplate>",
  "method":"POST",
  "contentType":"application/vnd.com.nsn.cumulocity.event+json",
  "placeholder":"%",
  "resourceUri":"/event/events",
  "templateString":
"{\"source\":{\"id\":\"%\"},\"type\":\"<type>\",\"time\":\"%\",\"text\":
\"<te xt>\", \"<name>\": \"%\" }",
  "values": [ "UNSIGNED", "DATE", "STRING" ]
}
```

Note: <time> can take either date value from device (if c8y_useServerTime is false) or NOW otherwise (then server time will be used on Cloud of Things platform). The actual coil and register values are logged as a property with the name of the coil or register and its corresponding value.

10.7. SEND MEASUREMENT

10.7.1. SMARTREST

```
<sendMeasurementTemplate>,<deviceId>,<time>,<register value>,...
```

10.7.2. SMARTREST REQUEST TEMPLATE

```
{
  "msgId":"<sendMeasurementTemplate>",
  "method":"POST",
  "contentType":"application/vnd.com.nsn.cumulocity.measurement+json",
  "placeholder":"%",
  "resourceUri":"/measurement/measurements",
  "templateString":
"{\"source\":{\"id\":\"%\"},\"time\":\"%\",\"type\":\"<type>\",\"<type>\
":{\"
<series>\":{\"value\":%,\"unit\":\"<unit>\"},...}}",
  "values": [ "UNSIGNED", "DATE", "NUMBER", ... ]
}
```

Note: <time> can take either date value from device (if c8y_useServerTime is false) or NOW otherwise (then server time will be used on Cloud of Things platform).

10.8. SETTING THE OPERATION STATUS TO EXECUTING

10.8.1. SMARTREST

```
102,<operationId>
```

10.8.2. SMARTREST REQUEST TEMPLATE

```
{
  "msgId":"102",
  "contentType":"application/vnd.com.nsn.cumulocity.operation+json",
  "method":"PUT",
  "placeholder":"%",
  "resourceUri":"/devicecontrol/operations/%",
  "templateString":"{\"status\":\"EXECUTING\"}",
  "values": [ "UNSIGNED" ]
}
```

10.9. SETTING THE OPERATION STATUS TO SUCCESSFUL

10.9.1. SMARTREST

```
103,<operationId>
```

10.9.2. SMARTREST REQUEST TEMPLATE

```
{
  "msgId":"103",
  "contentType":"application/vnd.com.nsn.cumulocity.operation+json",
  "method":"PUT",
  "placeholder":"%",
  "resourceUri":"/devicecontrol/operations/%",
  "templateString":"{\"status\":\"SUCCESSFUL\"}",
  "values":[ "UNSIGNED" ]
}
```

10.10. SETTING THE OPERATION STATUS TO FAILED

10.10.1. SMARTREST

```
104,<operationId>
```

10.10.2. SMARTREST REQUEST TEMPLATE

```
{
"contentType": "application/vnd.com.nsn.cumulocity.operation+json" ,
"method": "PUT" ,
"msgId": "104" ,
"placeholder": "%%" ,
"resourceUri": "/devicecontrol/operations/%%" ,
"templateString": "{\"status\":\"FAILED\"}" ,
"values":[
"UNSIGNED"
]
}
```

10.11. ADDING A DEVICE

10.11.1. SMARTREST

```
202,<line>,<operationId>,<deviceId>,<modbusId>,<modbusAddress>,<mod-
busDeviceType>,<modbusProtocol>
```

10.11.2. SMARTREST RESPONSE TEMPLATE

```
{
"condition": "$.c8y_ModbusDevice" ,
"msgId": "202" ,
"pattern":[
"$.id" ,
"$.deviceId" ,
"$.c8y_ModbusDevice.id" ,
"$.c8y_ModbusDevice.address" ,
"$.c8y_ModbusDevice.type" ,
"$.c8y_ModbusDevice.protocol"
]
}
```

10.11.3. CANOPEN

```
{
"condition": "$.c8y_CANopenAddDevice" ,
"msgId": "202" ,
"pattern": [
"$.id",
"$.deviceId",
"$.c8y_CANopenAddDevice.id",
"$.c8y_CANopenAddDevice.nodeId",
"$.c8y_CANopenAddDevice.type"
 ]
}
```

10.12. REMOVING A DEVICE

In CANopen protocol, there is also a device remove operation, which is sent to the terminal when a CANopen child device is deleted.

10.12.1. SMARTREST TEMPLATE

```
204,<line>,<operationId>,<deviceId>,<nodeId>
```

10.12.2.SMARTREST RESPONSE TEMPLATE

```
{
"condition": "$.c8y_CANopenRemoveDevice" ,
"msgId": "204" ,
"pattern": [
"$.id" ,
"$.deviceId" ,
"$.c8y_CANopenDevice.nodeId"
]
}
```

11. CHANGE LOG

11.1. FIELDBUS2 MODEL UPDATES

Register Definition: Add “multiplier” and “divisor” to be able to handle arbitrary scaling. To calculate the final value, multiply the register value by “multiplier”, divide it by “divisor” and shift it by “decimal places”. (Three values are used for scaling since most terminals do not support floating point operations.)

Device type definition:

- c8y_useServerTime: determines whether SmartREST templates should use server time or device time (e.g. in measurement templates).

Modbus configuration for terminal:

- pollingRate: defines time interval in seconds how often terminal should poll child devices
- protocol: can take one of these values: ASCII, RTU, TCP.

Modbus child device:

- c8y_ModbusDevice.ipAddress: only if Modbus/TCP is used.

11.2. FIELDBUS3 MODEL UPDATES

- All registers should be associated with a unit and “unit” is moved from measurement mapping to register definition. (Only relevant for user interface.)
- A register has an additional boolean flag “signed”. The register is interpreted as a signed integer.
- Support for modelling multiple logical bit and integer values inside one physical register.
    - A register definition contains two additional values “startBit” and “noBits” (number of bits) that determine the subset of bits to read from the register.
    - Register definitions can be repeated for the same register, provided “startBit” is different. This results in multiple values in “c8y_RegisterStatus” listed for one physical register, ordered by start bit.
    - Bits are numbered according to the Modbus specification with MSB leftmost.
    - Since Modbus does not allow partial setting of registers, we assume that the operation “SetRegister” will reconstruct the full value of the physical register before sending the register to the terminal, ⟶ Do not combine on user interface side but send startBit und noBits along with the setregister command.

- Change of enumeration values from array to a map, interpreting the values as text strings. For coils, the map will contain:
  `“enumValues”: { “0”: “<False text>”, “1”: “<True text>” }`
  For registers, there can be arbitrary additional values, representing generic enumeration types. Maps with two values are visualized as switch. Maps with more than two values are visualized as text resp. as drop-down box, when the register is editable.

- A register that can be updated has additional values “min” and “max” that constrain the values that can be set in the user interface.
- Change history: Any change of a coil or register is logged as an event through “event mapping”.
- Category: Is used for subdividing the Modbus widget into sections of coils and register, for example, setpoint parameters, actual parameters, compressor values.

11.3. FIELDBUS4 MODEL UPDATES

- Added support for discrete inputs by adding the flag “input” to coil definitions.
- Added support for input register by adding the flag “input” to register definitions.
- Added support for multiple protocols by adding property “fieldbusType” to device types.
- Added support for Modbus RTU
    - Protocol moved from c8y_ModbusConfiguration fragment in terminal managed object to c8y_ModbusDevice fragment in modbus child managed object
- Added support for CAN protocol
    - Register number for CAN types are hexadecimal numbers prefixed with “0x”.
    - Added offset parameter to register definitions for conversion.

11.4. FIELDBUS5 MODEL UPDATES

- Increased number of bits up to 64 (register.noBits) for Modbus, overlapping registers are not allowed.
- Enabled "Little endian" option for Modbus (register.littleEndian), shown only when selected number of bits is larger than 8.