|  |  |
| --- | --- |
| Input Contribution | |
| Meeting ID\* | SDS#43 |
| Title:\* | TR-0043-Modbus-interworking-procedures |
| Source:\* | JaeSeung Song, KETI, [jssong@sejong.ac.kr](mailto:jssong@sejong.ac.kr)  Sherzod Elamanov, KETI, [selamanov@gmail.com](mailto:selamanov@gmail.com) |
| Date:\* | 2019-12-01 |
| Input related to\* | TR-0043 Modbus Interworking |
| Intended purpose of  document:\* | Decision  Discussion  Information  Other <specify> |
| Impacted other TS/TR(s) | N/A |
| Decision requested or recommendation:\* | Add Section 7 Modbus interworking procedures |
| Template Version: January 2017 (Do not modify) | |

**oneM2M Notice**

The document to which this cover statement is attached is submitted to oneM2M. Participation in, or attendance at, any activity of oneM2M, constitutes acceptance of and agreement to be bound by terms of the Working Procedures and the Partnership Agreement, including the Intellectual Property Rights (IPR) Principles Governing oneM2M Work found in Annex 1 of the Partnership Agreement.

# Introduction

This contribution proposes to add a new section for Modbus Interworking procedures.

### -----------------------Start of change 1-------------------------------------------

# 7 Possible Solutions for oneM2M and Modbus Interworking

*This clause studies the possible solutions to realize oneM2M interworking with Modbus. Modbus-based devices can interwork with oneM2M system by usage of IPE that deploys on ASN, MN and IN, such as Modbus-based device connects to MN by IPE on MN. Resource mapping based on Modbus data model and operational procedure will be studied. Semantic method will also be consided in the solution.*

## 7.1 Exposure of Modbus Functions to the oneM2M System

### 7.1.1 Summary of Interworking Architecture for exposure of Modbus Functions

A Modbus-IPE that exposes Modbus Functions to the oneM2M System is responsible for the creation of oneM2M resources representing the exposed Modbus Functions on its Registrar CSE. A single Modbus-IPE may expose Modbus Functions provided by one or more Modbus devices to the oneM2M System. A high-level summary of the relationship of Modbus devices providing Modbus Functions to be exposed to the oneM2M System and the Modbus-IPE representing the exposed Modbus Functions is depicted in figure 7.1.1-1.



**Figure 7.1.1-1 Exposure of Modbus Functions to the oneM2M System via Modbus-IPE**

In the oneM2M System, Modbus devices are designed according to the oneM2M Smart Device Template (SDT) described in TS-0023. SDT offers a generic and flexible modeling structure for non-oneM2M devices including Modbus devices.

## 7.1.2 Registration

#### 7.1.2.1 <*AE*> resource representing a Modbus-IPE

The first step to support the Modbus interworking with the oneM2M System is to register a Modbus-IPE to its Registrar CSE as an <*AE*> resource. When the Modbus-IPE completes its registration with the Registrar CSE by initiating an <*AE*> Create request, an <*AE*> resource representing that Modbus-IPE is created as a result of the registration. This resource is a parent for <*flexContainer*> resource specializations representing Modbus devices connected to an associated Modbus Master. These devices are modelled using an SDT Device (details are described in the next section). Figure 7.1.2.1-1 shows an example resource tree structure of the *Modbus\_IPE* <*AE*> resource. *Modbus\_IPE* has a [*deviceThermometer*] as a child resource, which represents a thermometer Modbus device.

Modbus\_IPE

[deviceThermometer]

<subscriptions>

0..n

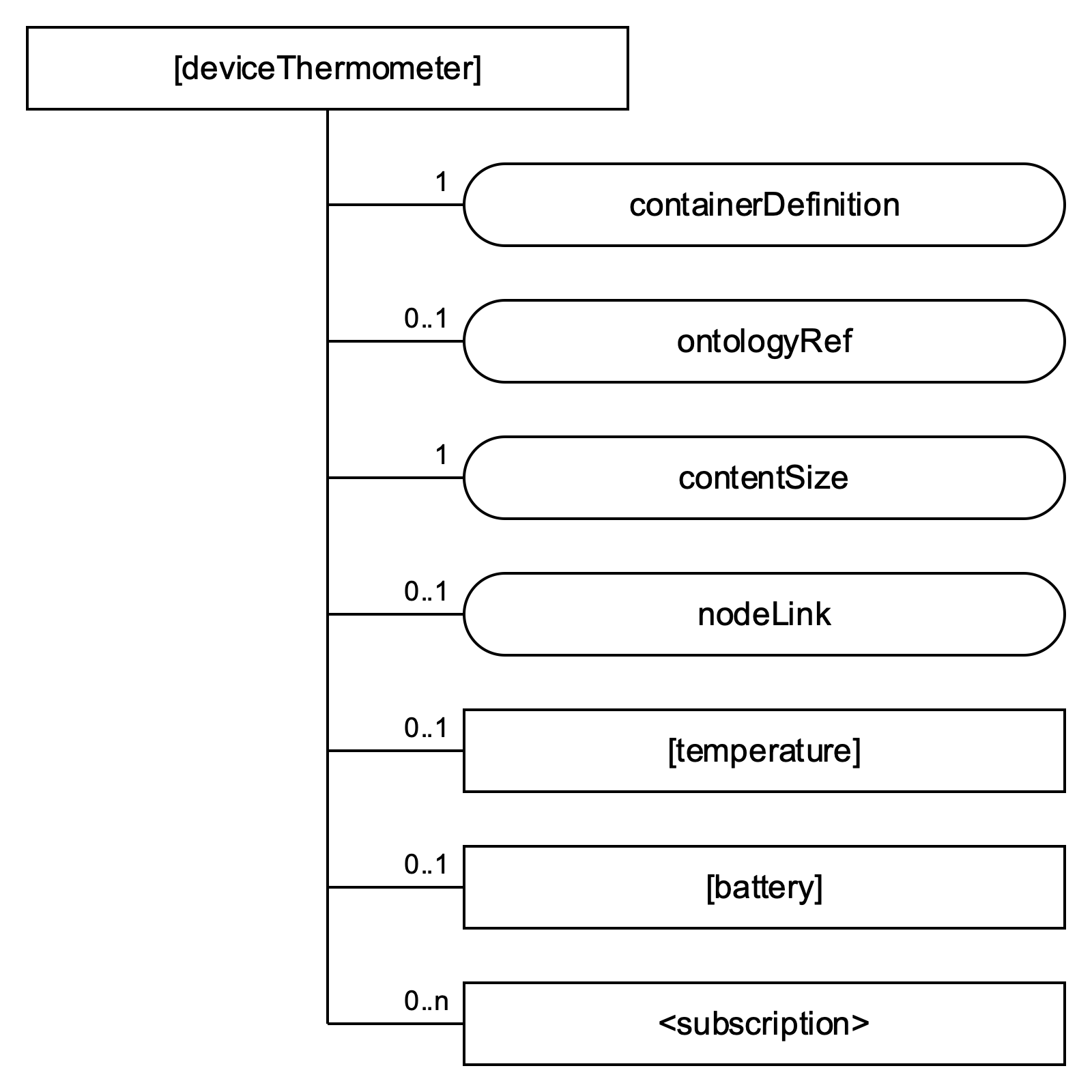
0..n

**Figure 7.1.2.1-1 *<AE>* resource representing a Modbus-IPE**

#### 7.1.2.2 deviceThermometer as a <*flexContainer*> resource specialization representing a Modbus Device

Modbus devices shall be modelled as SDT Devices. Mapping of the SDT Device model to oneM2M resources is performed according to the general mapping procedure described in clause 6.2.2 of TS-0023. A SDT Device component is mapped to a specialization of a <*flexContainer*> resource with an associated 'DeviceClass ID' (e.g. "org.onem2m.home.device.tv") *containerDefinition* attribute.

Figure 7.1.2.2-1 shows an example of a Modbus device:*[deviceThermometer],* which is modelled as *a <flexContainer>* resource specialization derived from the corresponding SDT Device component. The model of *[deviceThermometer]* follows the schema described in clause 5.5.45 of TS-0023.



**Figure 7.1.2.2-1 *[deviceThermometer]* resource representing a Modbus Slave**

#### 7.1.2.3 Defining SDT DataPoints based on a Modbus device’s register information

In order to enable interworking, a mapping between a Modbus device’s registers and SDT DataPoints is defined. Every Modbus register has the following properties: *slave id*, *register type*, *address*, *length*. The information of these registers are typically provided by a manufacturer in a device’s datasheet. Register type and length are used to define the following SDT DataPoint attributes: *DataType*, *writable* and *readable*. The rules to perform the mapping are shown in Table 7.1.2.3-1. A holding register and input register of length 2 can be mapped into either xs:integer or xs:float DataType depending on data context. As an example mapping, a coil register can be mapped to a DataPoint with DataType (xs:boolean), Redable (True), and Writable (True).

This information can be stored in an optional attribute called nodnProperties of the <flexContainer> specialization.

#### **Table 7.1.2.3-1 Mapping between Modbus register types and SDT Data points**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Modbus Register | | Mapping | SDT Data points | | |
| Modbus register type | Length | DataType | Readable | Writable |
| Coil (1 bit, Read-Write) | 1 (1 bit) | 🡪 | xs:boolean | True | True |
| Discrete Input (1 bit, Read-Only) | 1 (1 bit) | xs:boolean | True | False |
| Holding Register (16-bit, Read-Write) | 2 (4 bytes) | xs:integer / xs:float | True | True |
| Input Register (16-bit, Read-Only) | 2 (4 bytes) | xs:integer / xs:float | True | False |
| Holding Register (16-bit, Read-Write) | 1 (2 bytes) | xs:integer | True | True |
| Input Register (16-bit, Read-Only) | 1 (2 bytes) | xs:integer | True | False |
| Holding Register (16-bit, Read-Write) | 4 (8 bytes) | xs:double | True | True |
| Input Register (16-bit, Read-Only) | 4 (8 bytes) | xs:double | True | False |

#### 7.1.2.4 Temperature as a <*flexContainer*> resource specialization representing an SDT Module for a Modbus device

Depending on the functionalities of a target Modbus device, one or more ModuleClasses, which are generic ModuleClasses, defined in TS-0023 can be used to design a ModuleClass for the target Modbus device. The derived ModuleClass describes all functional capabilities of the target Mobus device.

For the [*deviceThermometer*] example described in clause 7.1.2.2, two ModuleClasses, Temperature (see clause 5.3.76 in TS-0023) and Battery (see clause 5.3.10 in TS-0023), can be used. DataPoints of a parent ModuleClass (in this example Thermometer) are created according to the mapping rule described in clause 7.1.2.3. The child ModuleClasses (in this case Temperature and Battery) inherit from those generic ModulesClasses.

ModuleClass is mapped into <*flexContainer*> resource specialization, for example Temperature, and its data points are mapped into *customAttributes* of that <*flexContainer*> resource specialization. However, those ModuleClasses do not consider interworking options with a non-oneM2M Device Nodes (noDN) such as Modbus devices. For that reason, a *nodnProperties* is added as *customAttribute* of a <*flexContainer*> resource which is mapped from an associated ModuleClass model.

The *nodnProperties* attribute stores one-to-one mappings in serialized string format (e.g. JSON) between each DataPoint and a Modbus register from which it is created. *nodnProperties* contains *slave id*, *register type*, *address*, and *length* attributes for each DataPoint as well as a protocol type (in this case Modbus). An example content of *nodnProperties* is shown on Figure **7.1.2.4-1.**

**Figure 7.1.2.4-1 An example contents of *noDNproperties***

{"currentTemperature": {

"slaveID": 1,

"registerType": "inputRegister", "address": "23", "length": 2

},

"targetTemperature": {

"slaveID": 1

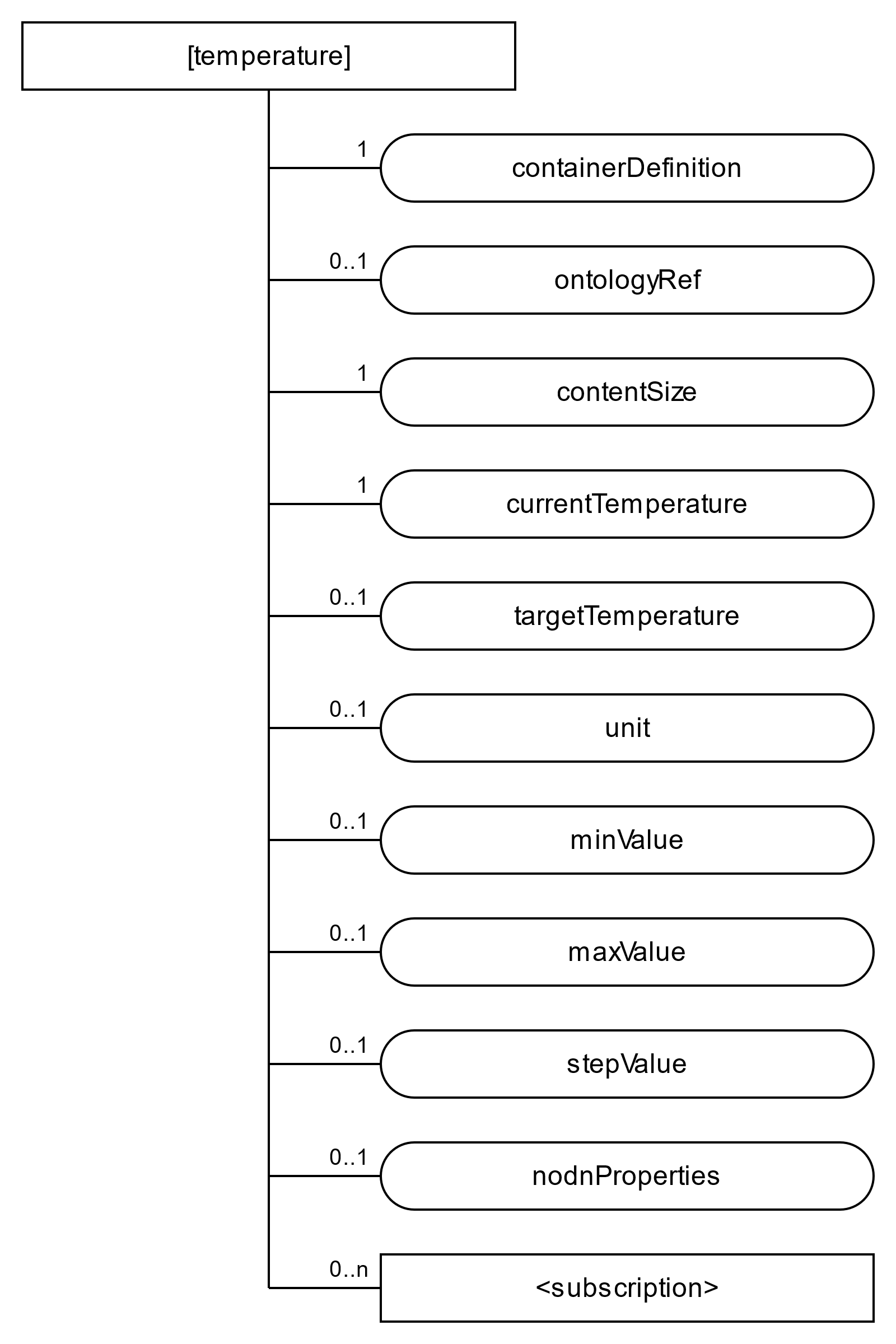
"registerType": "holdingRegister", "address": "25", "length": 2

},  
"unit": {…},

…

}

*<subscription>* resources are created for receiving notifications on <*flexContainer*> resource updates and shall be of type blocking subscription, that is *notificationEventType* attribute shall have a value "G" (see clause 9.6.8 in TS-0001). The blocking type of <*subscription*> resource ensures that a notification reaches its destination and no other UPDATE or DELETE operations are processed until the UPDATE has completed. *notificationContentType* attribute shall be set to "all attributes" so that *nodnProperties* attribute of a parent <*flexContainer>* resource is included into notification message. A subset of attributes of the subscribed-to resource that are triggering a notification when modified can be specified in the *attribute* tag of the *notificationEventCriteria* attribute.

**Figure 7.1.2.4-2 shows an example of a Module, the specialization for TemperatureModuleClass*,* which isa *<flexContainer>* resource specialization derived from a corresponding SDT Module component. Here it is assumed that DataPoints of the *[Temperature]* resource specialization are derived as a result of the mapping procedures described in clause 7.1.2.3 and DataPoints are obtained as in clause 5.3.76 of TS-0023. *nodnProperties* [*customAttribute*] is added to support Modbus interworking.

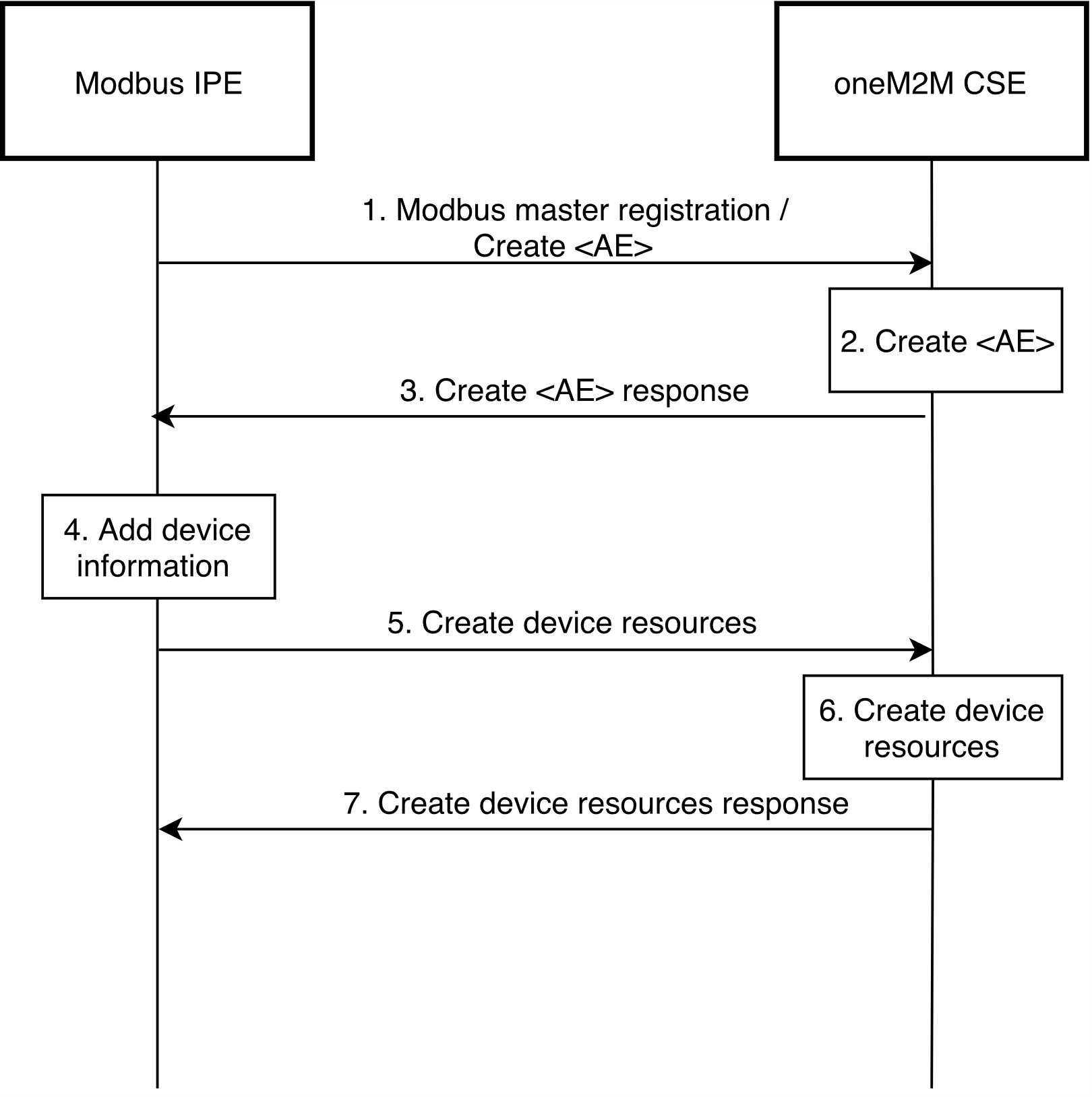
**Figure 7.1.2.4-2** ***[Temperature]* as a <*flexContainer*> resource specialization representing an SDT Module for Modbus device**

#### 7.1.2.5 Modbus device registration call flow

Figure 7.1.2.5-1 shows the device registration call flow.

1. Modbus IPE sends a Create *<AE>* request to a Hosting CSE to register the Modbus-IPE (see clause 7.4.5.2.1 in TS-0004). The request must specify *App-ID* and *requestReachabily* attributes of the to be created <*AE*> resource. Other <*AE*> attribtes are optional.
2. After verifying the privileges and the given attributes, the Hosting CSE creates the <*AE*> resource.
3. Hosting CSE responds with the successful result of *<AE>* resource creation, otherwise it responds with an error.
4. Modbus devices are registered at Modbus IPE, in particular Modbus interworking information (slave id, registers type, address, length) are defined in accordance with provided device datasheet.
5. Modbus IPE sends corresponding requests to a CSE to create resources as described in sections 7.1.2.1 - 7.1.2.3. For all <*flexContainer*> resources, the *containerDefinition* attribute is mandatory*.* The *contentSize* attribute is calculated by Hosting CSE. *CustomAttributes* must be specified if they are mandatory for that <*flexContainer*>. Each resource creation is originated by Modbus-IPE in a separate request for each resource.

For the presented above thermometerexample, *[deviceThermometer], [temperature]* and *[battery]* as child resources of *[deviceThermometer],* and <*subscription*> resources for *[temperature]* and *[battery]* shall be created.

1. After verifying the privileges and the given attributes, the Hosting CSE creates each resource.
2. ****Hosting CSE responds with the successful result for each created resource, otherwise it responds with an error.

**Figure 7.1.2.5-1 Device registration call flow**

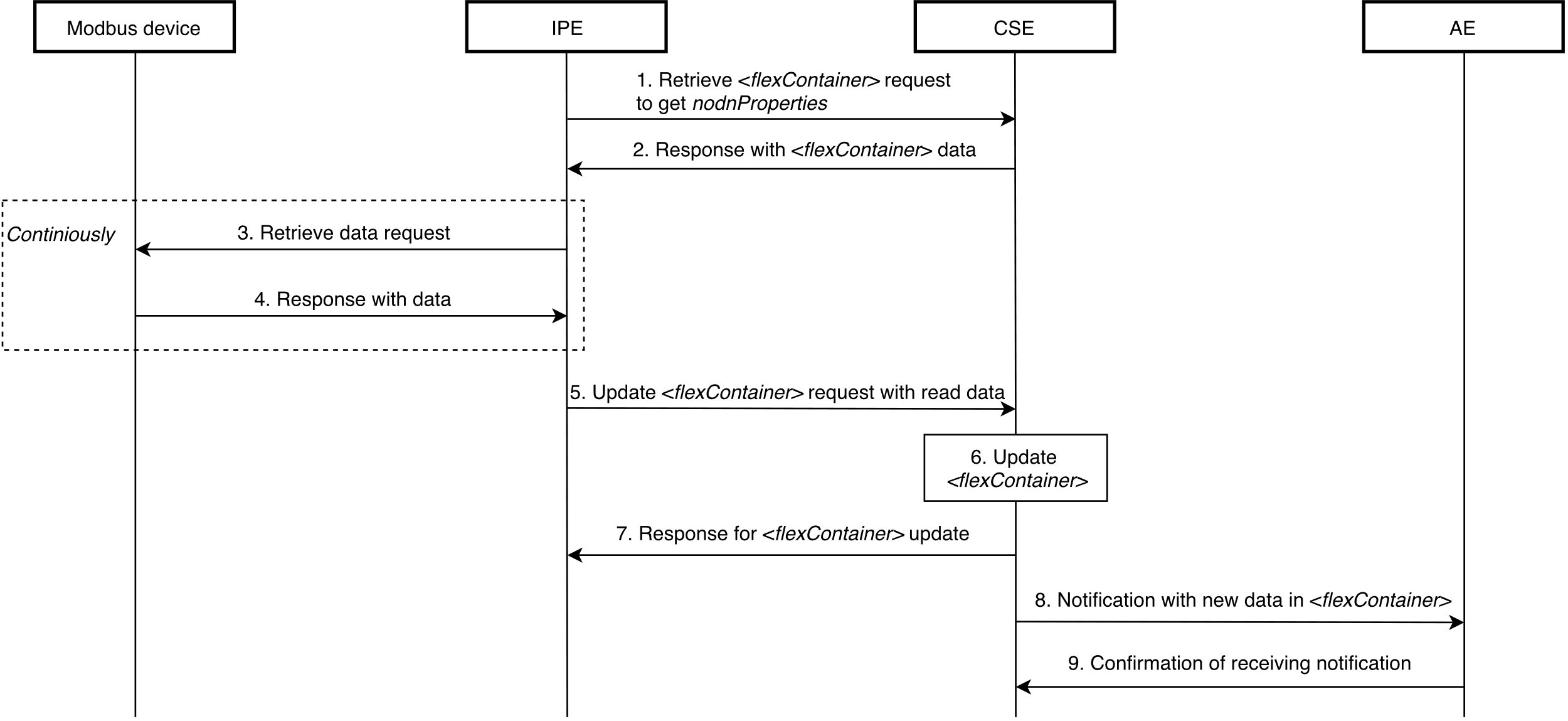
## 7.1.3 Retrieve data from a Modbus device

Suppose a scenario when current readings of a Modbus device need to be displayed at an AE application and Modbus-IPE continuously monitors a Modbus device and uploads that data to a CSE hosted on a server in the network. Inititially, AE shall be subscribed to the <*flexContainer*> resource, which is a specialization of some SDT module for a Modbus device (e.g. Temperature, see clause 7.1.2.4), using a <*subscription*> resource (*notificationEventType A,* see clause 9.6.8 in TS-0001). The following steps described in the Figure 7.1.3-1 shall be performed for this scenario:

1. Modbus IPE sends a retrieve <*flexContainer*> request to a hosting CSE. This <*flexContainer*> resource is a specialization of some Modbus module and contains *nodnProperties* attribute.
2. Hosting CSE responds to the retrieve request with <*flexContainer*> data that includes *nodnProperties*.
3. Modbus IPE uses information stored in *nodnProperties* to compose Modbus read request. The function code can be identified from a register type as in the Table 7.1.3-1. Slave id, address and length should written in correspong message fields. After theModbus message is composed, the Modbus IPE sends this message to Modbus device.

**Table 7.1.3-1 Register type to function code mapping for Modbus read request**

|  |  |
| --- | --- |
| **Register type** | **Function code** |
| Coil | 01 |
| Discrete input | 02 |
| Holding register | 03 |
| Input register | 04 |

1. Modbus device responds with requested data.
2. Modbus IPE sends an update <*flexContainer*> request (see clause 7.4.37.2.3 in TS-0004). The request body specifies the *customAttributes* to be updated and their new values read from Modbus device.
3. After verifying the privileges and the given attributes, the hosting CSE updates <*flexContainer*> resource.
4. The hosting CSE responds with updated <*flexContainer*> data after successful update to Modbus IPE, otherwise it responds with an error.
5. The hosting CSE sends a notification for <*flexContainer*> resource update to the AE (see clause 7.5.1.2.2 in TS-0004).
6. The AE sends a confirmation message about notification receiving to the hosting CSE (see clause 7.5.1.2.2 in TS-0004).

**Figure 7.1.3-1 Modus Slave Device monitoring call flow**

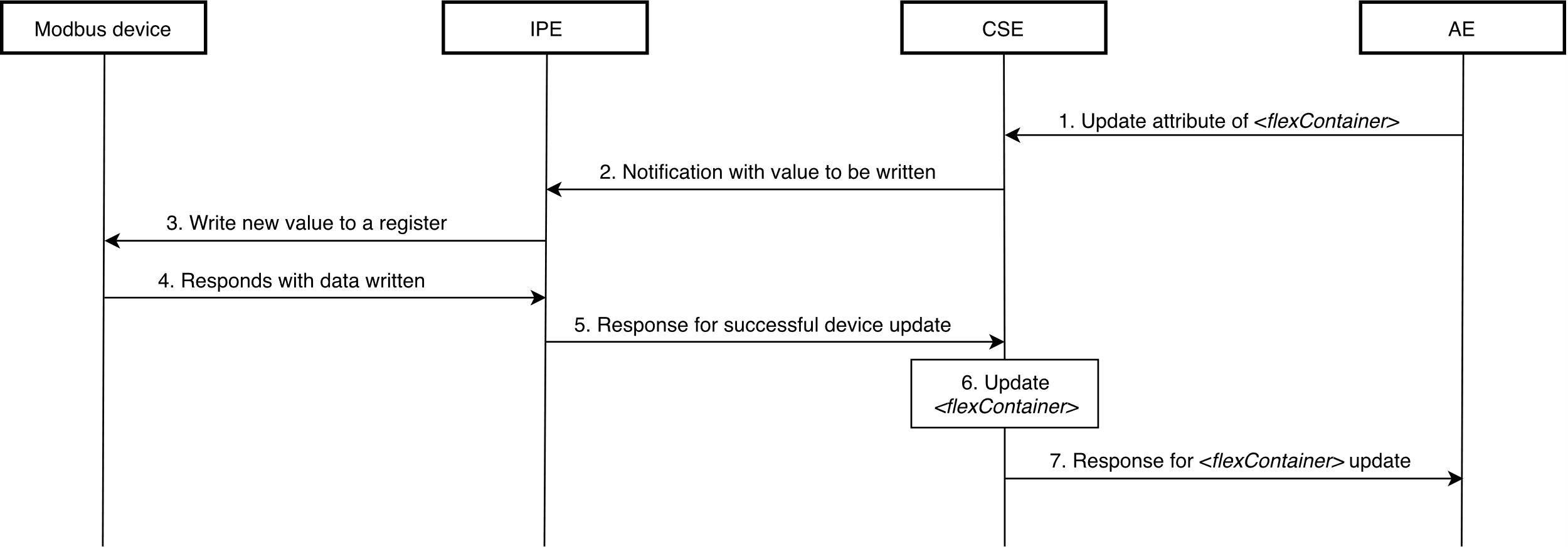
## 7.1.4 Write data to a Modbus Slave device

Suppose a scenario when it is required to update some value in a Modbus device through an AE application registered to a CSE. Inititially, the Modbus-IPE shall be subscribed to the <*flexContainer*> resource, which is a specialization of some SDT module for a Modbus device (e.g. Temperature, see clause 7.1.2.4), using a blocking type of <*subscription*> resource (*notificationEventType G,* see clause 9.6.8 in TS-0001). The following steps described in the Figure 7.1.4-1 shall be performed for this scenario:

1. In order to write data to a Modbus device from an AE, the AE sends a request to update specified *customAttributes* of the *<flexContainer>* resource which map to the Modbus Device (see clause 7.4.37.2.3 in TS-0004).
2. After verifying the privileges and the given attributes, the hosting CSE sends a notification for the received write request to the Modbus IPE (notification includes *nodnProperties*) and temporarily blocks the *<flexContainer>* resource for any UPDATE operations (see clause 7.5.1.2.2 in TS-0004).
3. Modbus IPE uses information stored in *nodnProperties* to compose Modbus write request. The function code to be used can be identified from a register type and length as in the Table 7.1.4-1. Slave id, address, and length should written in corresponding message fields. After Modbus message is composed Modbus IPE sends this message to Modbus device.

**Table 7.1.4-1 Register type and length to function code mapping for Modbus write request**

|  |  |  |
| --- | --- | --- |
| **Register type** | **Length > 1** | **Function code** |
| Coil | false | 05 |
| Coil | true | 0F |
| Holding register | false | 06 |
| Holding register | true | 10 |

1. Modbus device responds with written data to Modbus IPE.
2. Modbus IPE responds to the hosting CSE with successful device update message, otherwise responds with an error (see clause 7.5.1.2.2 in TS-0004).
3. If the device was updated successfully, the hosting CSE updates the *<flexContainer>* resource internally, otherwise discards the changes. The resource is unlocked for UPDATE operations.
4. The hosting CSE responds to AE with the result of UPDATE request.

**Figure 7.1.4-1 Writing to a Modbus Slave Device call flow**

### -----------------------End of change 1 -------------------------------------------

### -----------------------Start of change 2 -------------------------------------------

# X Conclusion

This TR investigates a mechanism how oneM2M application can access data stored in modbus devices via an interworking proxy. After an in-depth analysis of the Modbus protocol, an interworking mechanism via an interworking proxy using SDT 4.0 and <flexcontainer> resources. In particular, a *nodnProperties* attribute is introduced as a customAttribute of a <flexContainer> resource which is mapped from an associated Modbus ModuleClass model.

As the proposed interworking mechanism in this TR provides a proper solution for Modbus interworking, it is recommended to develop a normative standalone TS for supporting Modbus interworking based on the proposed mechanism.

### -----------------------End of change 2 -------------------------------------------