# 1 Scope

 The present document specifies the oneM2M and Modbus interworking technologies that enable Modbus devices and oneM2M entities produce/consume services.

The clause 5 defines the interworking architecture model that describes where the Modbus IPE is hosted and how the IPE is composed with.

The clause 6 defines the architecture aspects that mainly describes Modbus services to oneM2M resource mapping structure and rules. Furthermore, this explains the IPE registration and interworking procedures.

# 2 References

## 2.1 Normative references

References are either specific (identified by date of publication and/or edition number or version number) or non‑specific. For specific references, only the cited version applies. For non-specific references, the latest version of the reference document (including any amendments) applies.

The following referenced documents are necessary for the application of the present document.

[1] oneM2M TS-0001: "Functional Architecture".

[2] oneM2M TS-0004: "Core Protocol".

[3] oneM2M TS-0023: "Home Appliances Information Model and Mapping"

## 2.2 Informative references

References are either specific (identified by date of publication and/or edition number or version number) or non‑specific. For specific references, only the cited version applies. For non-specific references, the latest version of the reference document (including any amendments) applies.

The following referenced documents are not necessary for the application of the present document but they assist the user with regard to a particular subject area.

[i.1] oneM2M Drafting Rules.

NOTE: Available at <http://www.onem2m.org/images/files/oneM2M-Drafting-Rules.pdf>.

[i.2] Modbus website

NOTE: Available at <http://www.modbus.org/>

[i.3] Modbus\_Application\_Protocol\_V1\_1b3, Modbus Organization

[i.4] Modbus\_Messaging\_Implementation\_Guide\_V1\_0b, Modbus Organization

[i.5] Modbus\_over\_serial\_line\_V1\_02, Modbus Organization

[i.6] oneM2M TR-0009: "Protocol\_Analysis".

[i.8] oneM2M TR-0043: "Modbus Interworking".

# 3 Definitions and abbreviations

## 3.1 Definitions

For the purposes of the present document, the following terms and definitions apply:

**Modbus Slave/Device:** a peripheral device that provides a Modbus interface and responds by supplying the requested data to the master, or by taking the action requested in the query.

**Modbus Master:** a software running on a computer or a server as a host to access Modbus Slaves by issuing unicast requests.

## 3.2 Abbreviations

For the purposes of the present document, the following abbreviations apply:

AE Application Entity

CSE Common Services Entity

CSV Comma-Separated Values

IPE Interworking Proxy Entity

URI Uniform Resource Identifier

SDT Smart Device Template

XML eXtensible Markup Language

# 5 Architecture Model

## 5.1 Reference model

The architecture model followed in the present document is based on the architecture model in oneM2M TS‑0001 [1] that describes how interworking between oneM2M CSEs and non-oneM2M systems using specialized Interworking Proxy application Entities (IPE). The present document describes the Modbus IPE that supports the following reference model.

Modbus device

CSE

Modbus Protocol

Modbus IPE

Mca

Mca

ASN/MN/IN

CSE

Mcc/Mcc’

MN/IN

AE

AE

Mca

Figure 5.1-1: Modbus interworking reference model

## 5.2 Composition of the IPE

The Modbus IPE consists of AE and Modbus Master. To provide the interworking functions to other oneM2M entities, the IPE registers to a CSE and communicate with Modbus devices using Modbus protocol. The IPE registration is mandatory in oneM2M systems which is not defined in AllJoyn system. Modbus discovery and session establishment are needed for the IPE to communicate with other Modbus applications. A single Modbus IPE may expose Modbus functions provided by one or more Modbus devices to the oneM2M System.

**Figure 5.2-1: Composition of Modbus-IPE**

# 6 Architecture Aspects

## 6.1 Introduction

The present document specifies the functions for Modbus interworking in the following aspects:

* oneM2M resource mapping structure;
* Modbus IPE registration;
* Modbus service mapping;
* Modbus interworking procedures.

## 6.2 oneM2M resource mapping structure

6.2.1 Introduction

In this clause, the overall resource mapping structure for exposing service between Modbus devices and oneM2M entities is introduced. Firstly, Modbus devices are modelled according to the SDT described in TS-0023 [3]. The SDT offers a generic and flexible modeling structure for describing functionalities of non-oneM2M devices including Modbus devices. After the SDT schemas of the Modbus devices are created, they are mapped to oneM2M resources.

6.2.2 Mapping Modbus devices into SDT schemas

Each Modbus device shall be modelled as Device component. The Modules of the Device shall be created according to the functionality of the Modbus device as defined in the TS-0023.

For representing data objects of a Modbus device, the 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 is 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*, *readable,* and *optional*. The rules to perform the mapping are shown in Table 6.2.2-1. 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), Readable (True), and Writable (True). The *optional* attribute depends on a Modbus device and application logic and is supposed to be defined by the system integrator.

**Table 6.2.2-1 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 |

6.2.3 Mapping SDT schemas into oneM2M resources

The mapping of all SDT components follows the mapping procedure defined in clause 6.2 of TS-0023 [3]. For example, the ModuleClass models shall be mapped to the specializations of <flexContainer> resource and its DataPoints to customAttrubutes of corresponding <flexContainer> specialization. However, the SDT schemas do not consider interworking options with non-oneM2M Device Nodes (noDN) such as Modbus devices. For that reason, a *nodnProperties* attribute shall be added as a customAttributeof a <*flexContainer*> resource specialization which is mapped from an associated ModuleClass model.

The *nodnProperties* attribute stores one-to-one mappings in CSV string format between each customAttribute of <*flexContainer*> resource specialization and a Modbus register with which it is associated. Each line in the *nodnProperties* shall contain the name of customAttribute and associated Modbus register properties (*slave id*, *register type*, *address*, *length*). The order they are aligned is following: *customAttribute name, slave id*, *register type*, *address*, *length.* The *nodnProperties* shall have one record per line and each property separated by a comma. Table for 6.2.3-1 shows the detailed information on the fields of *nodnProperties* attribute.

An example oneM2M resource schema including *nodnProperties* is provided in Annex B, Figure B.2.2-2.

**Table 6.2.3-1 Fields of *nodnProperties* attribute**

|  |  |  |
| --- | --- | --- |
| **Field name** | **Type** | **Description** |
| customAttribute name | String | Name of customAttribute |
| slave id | Integer | Slave id of Modbus device |
| register type | Enumeration | One of 4 register types (see Table 6.2.3-1) |
| address | Integer | Address of the first register associated with a variable |
| length | Integer | Number of registers an associated variable occupies |

**Table 6.2.3-2 Interpretation of register type**

|  |  |
| --- | --- |
| **Value** | **Interpretation** |
| 1 | Coil |
| 2 | Discrete input |
| 3 | Holding register |
| 4 | Input register |

## 6.3 Modbus IPE registration

Figure 6.3-1 shows the device registration call flow.

1. The IPE requests to create an <AE> resource on the Hosting CSE to register the Modbus master collocated on the IPE.
2. The Hosting CSE evaluates the request, performs the appropriate checks, and creates the <AE> resource.Hosting CSE responds with the successful result of *<AE>* resource creation, otherwise it responds with an error.
3. 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.
4. Modbus IPE sends corresponding requests to a CSE to create resources which were from SDT schemas as described in the section 6.2.3. 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.
5. After verifying the privileges and the given attributes, the Hosting CSE creates each resource.
6. Hosting CSE responds with the successful result for each created resource, otherwise it responds with an error.



**Figure 6.3-1 Device registration call flow**

## 6.4 Modbus service mapping

The Modbus devices can accept either read or write requests from the Master. The operation to be executed is identified from the function code of a Modbus message. Therefore, the IPE needs to be able to map the oneM2M messages to Modbus messages with the appropriate function code. The function code is identified from register type of the register to be read for the read requests and from a tuple of register type and number of registers to be written (length) for

For the read requests, the IPE maps the register type of the register to be read to the function code according to the Table 6.4-1. For the write requests, the IPE maps the tuple of register type and the number of registers to be written (length) to the function code according to the Table 6.4-2. Both the register type and the length along with other Modbus data needed to construct the Modbus message can be retrieved from the *nodnProperties* customAttribute of a <*flexContainer*> specialization derived from a ModuleClass.

**Table 6.4-1 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 |

**Table 6.4-2 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 |

## 6.5 Modbus interworking procedures

6.5.1 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. Initially, AE shall be subscribed to the <*flexContainer*> resource, which is a specialization of some SDT module for a Modbus device, using a <*subscription*> resource (*notificationEventType A,* see clause 9.6.8 in TS-0001). The following steps described in the Figure 6.5.1-1 shall be performed for this scenario:

1. The Modbus IPE sends a retrieve <*flexContainer*> request to the hosting CSE. This <*flexContainer*> resource is a specialization of some Modbus module and contains *nodnProperties* attribute.
2. The Hosting CSE responds to the retrieve request with <*flexContainer*> data that includes *nodnProperties*.
3. The 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 6.4-1. Slave id, address and length should be written in corresponding message fields. After the Modbus message is composed, the Modbus IPE sends this message to Modbus device.
4. The Modbus device responds with requested data.
5. The 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.
6. After verifying the privileges and the given attributes, the hosting CSE updates <*flexContainer*> resource.
7. The hosting CSE responds with updated <*flexContainer*> data after successful update to the Modbus IPE, otherwise it responds with an error.
8. The hosting CSE sends a notification for <*flexContainer*> resource update to the AE (see clause 7.5.1.2.2 in TS-0004).
9. The AE sends a confirmation message about notification receiving to the hosting CSE (see clause 7.5.1.2.2 in TS-0004).



**Figure 6.5.1-1 Modus Slave Device monitoring call flow**

## 6.5.2 Write data to a Modbus device

Suppose a scenario when it is required to update some value in a Modbus device through an AE application registered to a CSE. Initially, the Modbus IPE shall be subscribed to the <*flexContainer*> resource, which is a specialization of some SDT module for a Modbus device, using a blocking type of <*subscription*> resource (*notificationEventType G,* see clause 9.6.8 in TS-0001). The following steps described in the Figure 6.5.2-1 shall be performed for this scenario:

1. In order to write data to a Modbus device from the 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. The 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 6.5-2. Slave id, address, and length should be written in corresponding message fields. After the Modbus message is composed the Modbus IPE sends this message to Modbus device.
4. The Modbus device responds with written data to the Modbus IPE.
5. The 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).
6. 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.
7. The hosting CSE responds to the AE with the result of the UPDATE request.

**Figure 6.5.2-1 Writing to a Modbus Slave Device call flow**

Annex A (Informative):
Introduction to Modbus

# A.1 Background

Modbus was first introduced by Modicon® (now part of Schneider Electric®) for process control systems. It is used to establish master-slave/client-server communication between intelligent devices and sensors and instruments. It is a de facto standard, truly open and the most widely used network protocol in the industrial manufacturing environment.

Modbus is easy to deploy and maintain, and used across a wide range of industries. It is also an ideal protocol for remote terminal unit (RTU) applications where wireless communication is required. Modbus is not only an industrial protocol. Building, infrastructure, transportation and energy applications also make use of its benefits.

Originally, Modbus was implemented as an application level protocol intended to transfer data over serial port, it has expanded to include implementations over serial, TCP/IP, and UDP. Today, it is a common protocol used by countless devices for simple, reliable, and efficient communication across a variety of networks. Modbus was designed as a request-response protocol with a flexible data and function model that are part of the reason it is still in use today. In addition, support for the simple and elegant structure of Modbus continues to grow.

# A.2 Architecture and protocol stack

The Modbus protocol follows a master and slave architecture where a master transmits a request to a slave and waits for the response (as shown in the Figure A.2-1). This architecture gives the master full control over the flow of information, which has benefits on older multidrop serial networks. Even on modern TCP/IP networks, it gives the master a high degree of control over slave behavior, which is helpful in some designs.



Figure A.2-1 The Master-Slave, Request-Response Relationship of Modbus device

The Modbus protocol allows an easy communication within all types of network (as shown in Figure A.2-2). Every type of devices (such as PLC, Driver, Motion control, I/O Device…) can use Modbus protocol to initiate a remote operation.

The same communication can be done as well on serial line as on an Ethernet TCP/IP networks. Gateways allow a communication between several types of buses or network using the Modbus protocol.



Figure A.2-2 Modbus Network Architecture

There are many variants of Modbus protocols:

* Modbus RTU — This is used in serial communication & makes use of a compact, binary representation of the data for protocol communication. Modbus RTU is the most common implementation available for Modbus. A Modbus RTU message must be transmitted continuously without inter-character hesitations.
* Modbus ASCII — This is used in serial communication and makes use of ASCII characters for protocol communication.
* Modbus TCP/IP or Modbus TCP — This is a Modbus variant used for communications over TCP/IP networks. It does not require a checksum calculation as lower layers already provide checksum protection.
* Modbus over TCP/IP or Modbus over TCP or Modbus RTU/IP — This is a Modbus variant that differs fromModbus TCP in that a checksum is included in the payload as with Modbus RTU.
* Modbus over UDP — Some have experimented with using Modbus over UDP on IP networks, which removes the overheads required for TCP.
* Modbus Plus (Modbus+, MB+ or MBP) — Modbus Plus is proprietary to Schneider Electric® and unlike the other variants, it supports peer-to-peer communications between multiple masters. It requires a dedicated co-processor to handle fast HDLC-like token rotation. It uses twisted pair at 1 Mbit/s and includes transformer isolation at each node, which makes it transition/edge triggered instead of voltage/level triggered.

At present, Modbus TCP is more efficient networking through the use of dedicated connections and identifiers for each request and response. Modbus RTU and Modbus ASCII are older serial ADU formats with the primary difference between the two being that RTU uses a compact binary representation while ASCII sends all requests as streams of ASCII characters.

The Modbus protocol defines a simple protocol data unit (PDU) independent of the underlying communication layers. The mapping of Modbus protocol on specific buses or network can introduce some additional fields on the application data unit (ADU). The Modbus frame is as shown in the Figure A.2-3.



Figure A.2-3 Modbus Frame

A Modbus frame or Modbus Application Data Unit (ADU) consists of the following:

* Additional address field: A field containing additional addresses used by the underlying communication protocol. It is 1 byte slave address over serial links (such as RS 232, RS 485). For Modbus TCP, it is called Modbus Application Protocol (MBAP) Header that include transaction identifier, protocol identifier, length and unit identifier.
* Modbus PDU: It is independent of underlying communication layer and consists of two parts: 1) 1-byte Function code to indicate identity of the requested service, 2) Variable length data field containing payload of the requested service. There are three types of Modbus PDUs: Modbus Request, Modbus Response and Modbus Exception.
* An optional error check field. Modbus TCP is not needed.

# A.3 Key feature

There are many devices and gateways that support Modbus, as it is a very simple protocol and convenient to transmit and understand. Specially, Modbus TCP/IP simply takes the Modbus instruction set and wraps TCP/IP around it. If you already have a Modbus driver and you understand Ethernet and TCP/IP sockets, you can have a driver up and running and talking to a PC in a few hours. Development costs are exceptionally low. Minimum hardware is required, and development is easy under any operating system. The following are key features of Modbus:

* **Communication mode**

Modbus uses master-slave/client-server communication mode, Master issues a unicast request and slave responds to that. In serial and MB+ networks, only the node assigned as the Master may initiate a command. On Ethernet, any device can send out a Modbus command, although usually only one master device does so. Modbus also supports broadcast mode where master’s request is sent to all the slaves but no slave responds to broadcast request.

* **Data model**

Modbus manages the access of data simply and flexibly. Modbus data are divided into four ranges, they are that these types of data can be provided/alterable by I/O system or an application program. In most cases, slaves store each type of data that it supports in separate memory, and limits the number of data elements that a master can access.

* **Function code**

There are three categories of Modbus Function codes, including Public Function codes, User-Defined Function codes and Reserved Function codes. Public Function codes can satisfy common operations, such as accessing data in device by reading and writing data model, and simply diagnosing device. Function code is flexibility that user can select and implement a function code by self-defining User-Defined Function codes according to service requirements.

* **Availability of many devices**

Interoperability among different vendors' devices and compatibility with a large installed base of Modbus-compatible devices makes Modbus an excellent choice.

# A.4 Data model

The Modbus standard defines bit-addressable and 16-bit word addressable input and output data items. Modbus bases its data model on a series of tables that have distinguishing characteristics. The four primary tables for data model are as following:

**Table A.4-1 Modbus data model table**

|  |  |  |  |
| --- | --- | --- | --- |
| Primary tables | Object type | Type of access | Comments |
| Discretes Input  | Single bit | Read-Only | This type of data can be provided by an I/O system, e.g. read the status of switch |
| Coils  | Single bit | Read-Write | This type of data can be alterable by an application program, e.g. switch on a transducer |
| Input Registers  | 16-bit word | Read-Only | This type of data can be provided by an I/O system, e.g. read temperature on a sensor |
| Holding Registers  | 16-bit word | Read-Write | This type of data can be alterable by an application, e.g. set value to a controller. |

There are two ways of organizing the data in device. Each device can have its own organization of the data according to its application. The figure A.4-1 below shows an example for data organization in a device having digital and analog, inputs and outputs. Data block (device application memory) is accessible with different Modbus functions, such as read coils, write holding registers. All the data elements handled via Modbus can be located in device application memory by reference numbers form 1to n. The pre-mapping between the Modbus data model and the device application is totally vendor device specific.



**Figure A.4-1 Implementation example of Modbus data model**

Annex B (Informative):
Resource mapping examples

# Introduction

The IPE constructs oneM2M resource tree on hosting CSE from the SDT schemas derived from the set of functionalities of Modbus devices.

The present clause gives an example of how to use the oneM2M resource tree to represent a Modbus device (i.e. Thermometer).

# Example for thermometer device

The present clause explains the creation process for an arbitrary thermometer device that communicates over Modbus. As the Modbus devices are firstly represented by SDT models, the SDT definition of the thermometer device described in the clause 5.5.45 of TS-0023 will be considered.

## B.2.1 Example for Device model 'deviceThermometer'

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. Figure B.2.1-1 shows an example of the [*deviceThermometer*], which is modelled as a <*flexContainer*> resource specialization derived from the corresponding SDT Device component.

Figure B.2.1-1: Structure of *[*deviceThermometer*]* resource

## B.2.2 Example for ModuleClass 'temperature'

The SDT model of the 'temperature' ModuleClass is described in the clause 5.3.76 of TS-0023. Assume the DataPoints of the 'temperature' ModuleClass are created according to the mapping rule described in clause 6.2.2.

Mapping of the SDT ModuleClass model to oneM2M resources is performed according to the general mapping procedure described in clause 6.2.3 of TS-0023. The 'temperature' ModuleClass is mapped into [*temperature*], a <*flexContainer*> resource specialization, and its data points are mapped into customAttributesof that <*flexContainer*> resource specialization; and *nodnProperties* customAttribute is added the [*temperature*] as described in the clause 6.2.3. The figure B.2.2-1 shows the structure of [*temperature*].

The example contents of *nodnProperties* is shown on Figure B.2.2-2**.**

Figure B.2.2-1: Structure of *[*temperature*]* resource

"currentTemperature",1,4,23,2

"targetTemperature",1,3,25,2

"unit",1,4,27,2

"minValue",1,4,29,2

"maxValue",1,4,31,2

"stepValue",1,4,33,2

Figure B.2.2-2: Example contents of *noDNproperties*