System Specifications

3

KNX IoT

10

KNX IoT Point API

5

Summary

This document is part of the KNX IoT Standard. This chapter describes the Point API, the device configuration, and the runtime interworking.

This document is a KNX Approved Standard.

Document updates

Version	Date	Modifications
0.1.0	2021.03.22	- Document creation.
1.0.0	2021.11.01	- Draft for Voting
1.0.0	2022.04.05	- Creation of the Approved Standard
1.1.0	2023.01.20	- Errata document for certification
1.1.0	2023.05.10	- Creation of the Approved Standard
1.1.0	2023.06.21	- Publication of the Approved Standard.

References

[01] Chapter 3/10/1 KNX IoT Introduction
[02] Chapter 3/10/2 KNX IoT Constants
[03] Chapter 3/10/3 KNX IoT Information Model
[04] Chapter 3/10/4 KNX IoT 3rd Party API
[05] Chapter 3/10/5 KNX IoT Point API
[06] Chapter 3/1/2 Glossary
[07] Chapter 7/20/1 Lighting Sensors
[08] Chapter 7/20/2 Lighting Actuators
[09] Chapter 3/5/2 Management Procedures
[10] Chapter 3/5/1 Resources
[11] Chapter 3/3/7 Application Layer
[12] Chapter 3/10/6 KNX IoT Router
[13] SPAKE2+, an Augmented PAKE (IETF Draft: draft-bar-cfrg-spake2plus-04)

Filename: 3_10_5 KNX IoT Point API.docx
Version: 1.1.0
Status: KNX Approved Standard
Save date: 2023.06.21
Number of pages: 164

Contents

1 KNX IoT Point API

1.1 Introduction

KNX IoT uses Internet Protocol (IP) suite standards for the transmission of KNX IoT application layer data across IP networks.

Physical media like Ethernet (IEEE 802.3), Wi-Fi (802.11), or WPAN (802.15.4) carry KNX IoT packets. These may contain unicast TCP or UDP frames or multicast UDP frame transmission of KNX IoT application data. KNX IoT application data is agnostic to the underlying communication layers. Hence, it is possible to send KNX IoT messages over non-IP transport bindings such as NB IoT, or BLE. However, this is out-of-scope of this specification.

A typical (IP-based) interworking infrastructure allows heterogeneous data link media to work seamlessly with each other. The Point API maps KNX AL data to a RESTful resource model, and CBOR/JSON-based data representation is used to communicate over IP. Using this API, a client can read/write values or subscribe to Point events (e.g., switch on/off). The KNX IoT Point API is based on the following building blocks:

1. **Discovery**

Discovery (of resources, including devices) can be made with unicast or multicast. Resource discovery in CoAP (CoRE) is accomplished using a "/.well-known/core" resource URI that returns a list of links about resources (e.g., *Functional Block* Properties) hosted by that server that matches filter attributes.

2. **S-Mode Messaging**

The S-Mode messaging uses a secure message-oriented communication pattern for group communication where a producer sends a message to notify consumers of a change in the domain. A tool, or rather a Management Client (MaC), configures group communication events via group tables.

3. **Point Read, Write, and Publish/Subscribe**

Parameter and diagnostic Properties are used for sensor, actuator, parameter, and diagnostic values, such as getting the current sensor value or setting a setpoint. They are addressed by URIs, can be directly accessed with the corresponding standard CoAP access method GET (read values), and can be manipulated with PUT/POST (write values). Additionally, also subscribing to Property values is possible.

4. **Security**

Group communication and access to the parameter and diagnostic Properties are secured by OSCORE or (D)TLS. OSCORE (application layer security) is used for S-Mode messaging, Point read, write, and publish/subscribe. (D)TLS (transport layer security) is mainly used for mutually authenticated pre-shared key distribution. The initial bootstrapping of pre-shared OSCORE keys and operational device certificates bases on an out-of-band (QR code, NFC, or BLE, etc.) authentication code as a proof of possession.

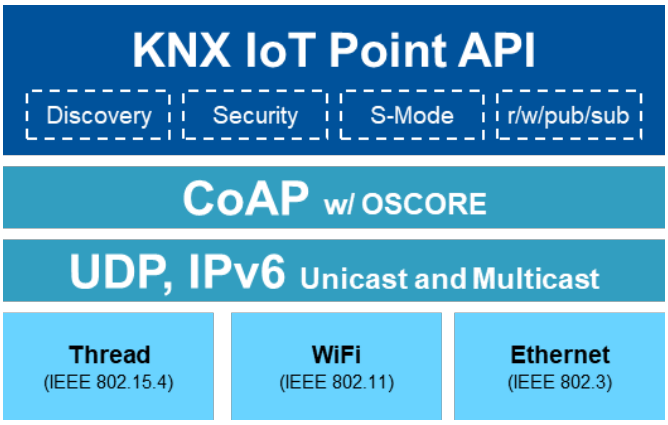


Figure 1 – KNX IoT Point API

1.2 System Entities

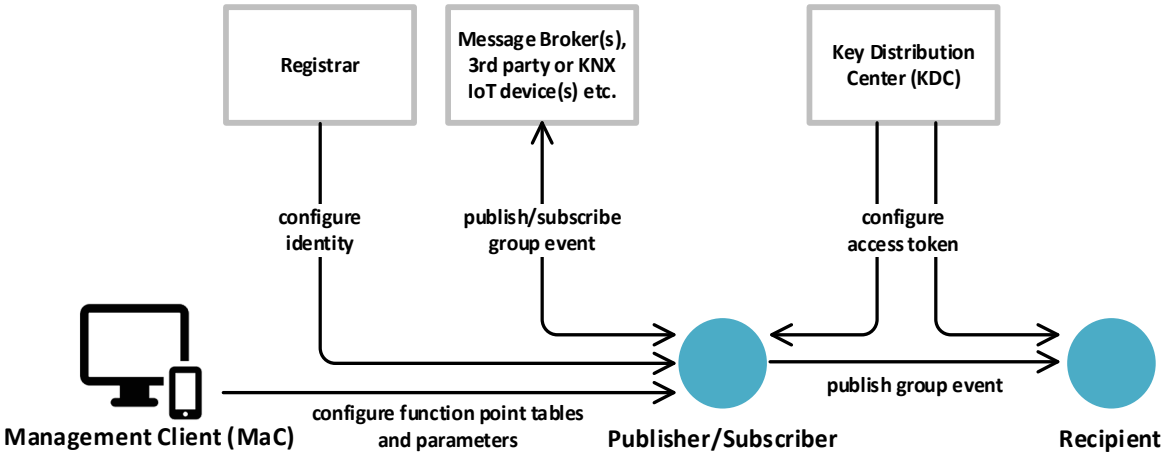


Figure 2 – System Entities

Some of the above-used terms are detailed underneath with additional KNX IoT Point API information. The general term definitions can be found in clause [02].

The **Key Distribution Center (KDC)** enables and enforces the authorized access of joining KNX IoT devices to access the related KNX Group Communication. Multiple installations can be associated with the same Authorization & Group Manager, which might be a service entity part of a MaC (Management Client) or an independent service.

The **Message Broker** is an intermediary that can provide data marshaling, routing, message translation, and persistence. It receives messages from a Publisher and delivers them to all related Recipients. A KNX Classic to KNX IoT Router (see also [12]) is an example of a Message Broker.

Publishers and **Subscribers** are KNX IoT devices that are loosely coupled and often do not know each other. Their primary relationship is that they are configured with the same *Group Address*. In KNX IoT:

- the Publisher sends Group Messages to Subscribers (e.g., via Message Broker). It represents a particular information source, for example, a light switch status;
- the Subscriber is the consumer of Group Messages from a Publisher. They are CoAP Clients but are Group Message Recipients;
- the Publishers and Subscribers use CoAP to communicate;
- A Recipient is a device that receives Group Messages from a Publisher. If the Recipient is not a Subscriber simultaneously, then the Recipient endpoint is a static configuration in the Publisher Function Point table.

The **Registrar** decides whether a new KNX IoT device can join the domain; it:

- might be a part of the MaC (Management Client), containing a mixture of Backlist rules, Whitelists (for known installed devices), and stateful tracking to protect the KNX System;
- subsequently configures domain CA certificates followed by the operational device certificate on KNX IoT devices.

1.3 Device Model

The following figure depicts the KNX IoT Device Model, which is, from a high-level perspective, the same as the classic KNX Device Model.

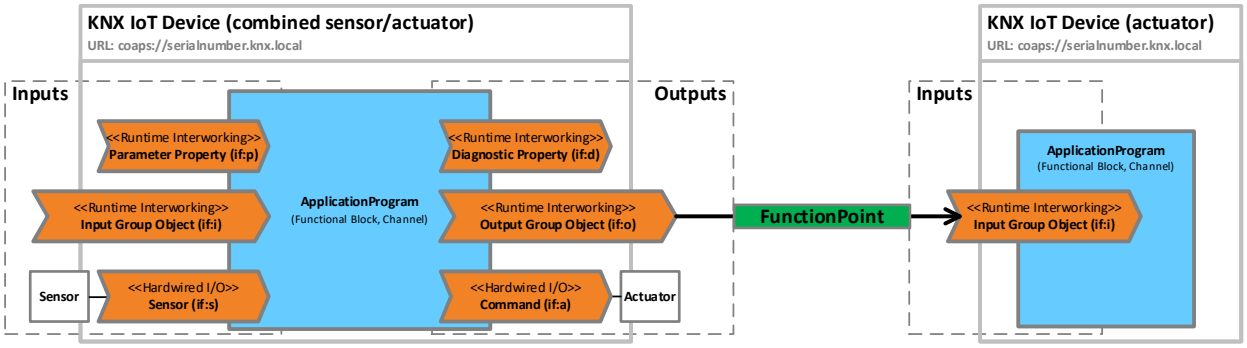


Figure 3 - KNX IoT Device Model

A KNX IoT device may have physical inputs and/or outputs.

- A physical input may be internal to the device or have an external sensor connected via a terminal block.
- A physical output may be internal to the device or have an external actuator connected via a terminal.

A KNX IoT device has at least one logical input and/or output. Such a logical input or output is called *Group Object (GO).* The *Group Object* has a unique identifier with reference to the device. When a *Group Address* is assigned to a *Group Object*, the Group Object becomes a member of that Function Point identified by the *Group Address*. A *Group Address* is the instantiation of a *Function Point* and is unique in an installation.

An input *Group Object* can be assigned to one or more *Function Points*.

An output *Group Object* can be assigned to one or more *Function Points* but can only send information to one *Function Point*.

For configuration purposes, a KNX IoT device may have parameters to determine specific behavior concerning the whole device or a device Channel. Each Parameter Property has a unique identifier with reference to the device.

For diagnostic purposes, a KNX IoT device may provide diagnostic information concerning the whole device or a device Channel. Each Diagnostic Property has a unique identifier with reference to the device.

Parameter and diagnostic Properties use the same communication mechanisms and are not differentiated, and both are named Property in the following.

A Group Object is a specialization of a Property. A Group Object is designed for runtime communication in a Brokerless or Broker-based system (see clause 2.3). A simple property is only designed and intended for configuration or diagnostic purposes, and it supports only simple point-to-point communication methods (read-, write- and subscription-commands).

A KNX IoT device may have *Group Objects* and parameter and diagnostic Properties that belong together. Where *Group Objects* and Properties belong together, these MAY be declared as a Channel.

A device may have Channels with different sets of Group Objects and Properties and/or Channels that have identical sets of Group Objects and Properties. Devices may have one or more Channels that are functionally identical and have the same type of Group Objects and Properties.

Apart from Channels, a KNX IoT device may have *Group Objects*, parameter, and diagnostic Properties that apply to the whole device. These are associated with the control function of the whole device.

If and how *Group Objects* and *Properties* are declared as a Channel is up to the manufacturer designing the device.

For certification purposes, a Channel may be declared to include at least one *Functional Block* describing the function of that Channel (see [03] "KNX Information Model").

The KNX Specification defines *Functional Blocks* for different applications. *Functional Blocks* encompass inputs and outputs as well as parameter properties with generic names and identifiers.

The *Functional Blocks* determine the Interworking between products of different manufacturers. More than one *Functional Block* type may be required to describe the functionality of a Channel.

A KNX IoT device has a unique identifier called *KNX Individual Address*. The *KNX Individual Address* is assigned and is unique with respect to the project.

A KNX IoT device SHALL have a globally unique identifier assigned by the manufacturer called *KNX Serial Number*.

The hardware (MAC-) or network (IP-) address of the device SHALL NOT be used as *KNX Individual Address*.

The following table summarizes the above-mentioned core elements and their associated unique identifiers.

Table 1 – Core elements and identifiers

Core element	Unique Identifier	
	Name	Scope
Device	KNX Serial Number	globally unique
	KNX Individual Address	unique within a project
Functional Block	Functional Block ID	unique per functionality
Function Point	Group Address	unique within a project
Group Object (GO)	GO identifier	unique per device
Parameter / Diagnostic Property	Property identifier	unique per device

1.4 Conventions used in this document

1.4.1 Requirements Language

The keywords "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", "MANDATORY", and "OPTIONAL" in this specification are to be interpreted as described in [RFC2119].

1.4.2 Conformance

For all resources, their expected request/ response document formats and content are not described to the full extent in this specification. As described in clause "Document Structure" in [01], whether an element is mandatory or optional can be found in the JSON-schema description as an electronic document.

The URL https://schema.knx.org allows retrieval of the most recent electronic documents of the KNX IoT Point API. Older versions are also available.

1.4.3 Number Format

All numbers in this document are assumed to be decimal unless indicated with a prefix. Hexadecimal numbers are prefixed with the designation "0x", and binary numbers are prefixed with the designation "0b". Binary numbers are defined as successive groups of 4 bits, separated by a space character from the most significant bit to the least significant bit (0b1111 1111). Byte strings are notated in one of the base encodings, without padding, enclosed in single quotes, prefixed by >h< for base16, >b32< for base32, >h32< for base32hex, >b64< for base64. For example, the byte string 0x12345678 could be written as h'12345678'.

1.4.4 Uniform Resource Identifiers

A Uniform Resource Identifier (URI) identifies a logical resource. This document uses the CoAP URL for link formats which comprises scheme, authority, path, and query values: scheme ":" ["//" authority] path ["?" query]. The authority part SHALL contain either an FQDN or an IP address.

The following link formats are used in this document:

Absolute Link with schema and authority: coap://{ip-address or fqdn}:{port}/base-path/path1?query

Absolute path that starts at the root path ("/"): /base-path/path?query

Relative path that starts after a base path (without "/" at the beginning): path?query

1.4.5 Uniform Resource Name

A Uniform Resource Name (URN) identifies a logical resource in a permanent way, even after that resource does not exist anymore. This document uses the URN format as defined in [RFC2141], which comprises the namespace identifier "knx" and namespace-specific identifiers: "urn:knx" [":" namespace-specific-identifier].

The following URN formats are used in this document:

Absolute URN with the namespace identifier: urn:knx:namespace-specific-identifier

Relative URN without namespace identifier (with ":" at the beginning): :namespace-specific-identifier

2 Point API Specification

2.1 Application Protocol

This version of the KNX IoT Point API specification solely considers the use of CoAP as an application protocol. The below specifications and examples, therefore, focus on CoAP. Indications for other application protocols may be added as these are added to this specification.

2.2 Overview

2.2.1 Common Data Model

The following clauses describe a standard data model and API for KNX IoT devices that represent a contract for the interworking between devices and, in addition, between devices and infrastructure (e.g., Management Client, Message Broker, etc.). For a semantic description of instances of the API (project export), KNX IoT adopts the W3C WoT interactions model for Points that exchange data via IP interworking infrastructure. The Thing Description [TD] is a central building block in the W3C Web of Things (WoT). The TD provides a framework for semantic metadata for the device itself, an interaction model based on WoT's PropertyAffordances, ActionAffordances, and EventAffordances paradigm, a semantic schema to make data models machine-processable and features for Web Linking to express relations among devices and Points. For example, the "/.well-known/core" URI is an ActionAffordance with a default entry point for discovery purposes, or a PropertyAffordance is usually mapped to a parameter and diagnostic Property. The TD is the glue between KNX IoT device data and the KNX domain model (see [03] "KNX Information Model").

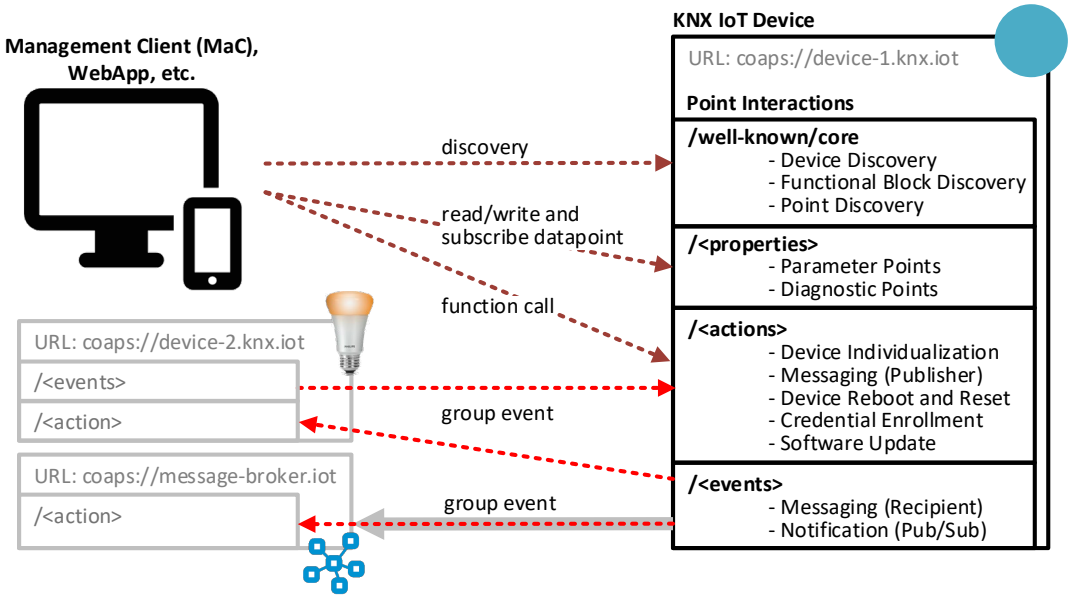


Figure 4 – Point Interactions

2.2.2 Application Layer Service Mapping

The KNX application layer provides many application services to the application process. Application processes in different devices interoperate by using services from the application layer. Depending on the communication mode, various application layer services are available. KNX IoT only takes over the core functionality of a subset of existing KNX application layer services. These service functionalities are mapped to REST calls which can be described as PropertyAffordances, ActionAffordances, or EventAffordances, for example, for a semantic project export or device descriptions (see [03] "KNX Information Model"). The following clause explains how existing KNX application layer services are mapped to service discovery, PropertyAffordances, ActionAffordances, and ventAffordances in KNX IoT.

The A_IndividualAddress_Read- and A_IndividualAddress_Write-services, for example, are used to find a device within a system and to change the device address. These services are used with broadcast communication, and the communication partner is not identified in this service., i.e., the partner device is enabling its *Programming Mode*, e.g., by pressing a programming button on that device. In KNX IoT, the service is split into a discovery ActionAffordance (discovery phase: DNS or GET coap://{ipv6-multicast}/.well-known/core?if=urn:knx:if.pm) and a second subsequent authorized ActionAffordance call to change the device address (POST coap://{ipv6-unicast}/.well-known/knx/ia).

PropertyAffordances expose the internal state of a device that can be directly accessed (read) and optionally manipulated (write) on parameter and diagnostic Properties. Devices can also choose to make parameter and diagnostic Properties subscribable (observe) by pushing the new state after a change. PUT and GET on Properties are used instead of the KNX Classic Data Property services, such as A_PropertyValue_Read or A_PropertyValue_Write.

ActionAffordances offer device functions, and these functions may manipulate the internal state of a device in a way that is impossible through setting Properties. Examples are changing internal state that is not exposed as PropertyAffordances (e.g., A Restart-PDU), changing multiple Properties, or changing Properties over time.

An EventAffordance describes event sources that asynchronously push messages. Hence, event-oriented runtime communication is mapped to EventAffordances in the sending device and ActionAffordances in the receiving device. Here not state, but state transitions (events) are communicated (e.g., "light switch ON"). Events may be triggered by an internal state change that is not exposed as PropertyAffordance, e.g., a push button event to switch from Off to On. POST is used for Group Objects instead of KNX Classic A_GroupValue services.

2.2.3 Application Protocol

This document uses CoAP [RFC7252] as a MANDATORY application protocol. CoAP is a reliable transport that can preserve packet ordering and handles message duplication. However, the Point API specification is not limited to CoAP. Some device profiles MAY implement a CoAP and an HTTP interface. This enables an HTTP client to access data on a KNX IoT device without implementing a CoAP communication stack. However, how an HTTP request is mapped to a CoAP request and how a CoAP response is mapped back to an HTTP response is out-of-scope of this document, but an HTTP implementation SHOULD follow the guidelines described in [RFC8075].

2.2.4 Content-Format

KNX IoT device resources SHALL support the application/link format, the application/octet-stream format, or the Concise Binary Object Representation (CBOR) format [RFC8949]. The CoAP Content-Format ([RFC8949], clause 9.4), used to interact with Points via a RESTful method, SHALL use the Internet media type [RFC6838] application/cbor ([RFC8949], clause 9.4). The application/link format is used to list resources such as Points, and octet/stream is used for file transfer.

KNX IoT resources MAY also support the JavaScript Object Notation (JSON) format [RFC8259].

The examples given in this document are shown in JSON or CBOR diagnostic notation for human readability ([RFC8949] clause 8).

In the absence of the CoAP Accept and Content-Format options in a CoAP request or a successful CoAP response, devices SHALL assume the default format specified for the corresponding device resource for each of those options.

This document uses the following CBOR notation (for major type 7, the minor types used in this document are a subset only of the specified minor types [RFC8949], clause 3.3):

348 **Table 2 – Used notations**

short notation	CBOR type	major type	minor type	Notes
unsigned	unsigned integer	0		The length depends on the value.
negative	negative integer	1		The length depends on the value.
byte string	byte string	2		The length is dynamic.
text	text string	3		The length is dynamic.
[]	Array	4		The length is dynamic, as needed by the Point, i.e., a B8 with only 2 bits used will be encoded as an array of bools with length 2. In case we have a homogeneous array of a specific type, we indicate this by adding the <type>*.
{}	Map	5		The length is dynamic.
<tag>(CBOR type)	Tagging	6		<tag> is a number.
<1>(unsigned)	date/time	0	1	Numerical representation of seconds relative to 1970-01-01T00:00Z in UTC time.
bool	false/true	7	20 / 21	
float	single precision float	7	26	IEEE754 32-bit float.

349 ## 2.3 System Design

350 ### 2.3.1 Events and Group Communication

351 KNX group communication uses a message-oriented communication pattern for point-to-multipoint
352 communication where a producer sends a message to notify consumers of a change in the domain. An
353 event is a message that informs various listeners about something that has happened. A key element of an
354 event is that the producer potentially does not care much about the response. Often it does not expect any
355 answer at all, or if there is a response that the producer does care about, it is indirect.

356 There are two concepts for distributing events in a message-oriented system: Message Broker-based
357 Publish–Subscribe (pub/sub) and brokerless transport layer multicast (or unicast) messaging pattern.
358 These concepts are described in the following clauses.

359 ### 2.3.2 Brokerless System

360 The KNX IoT point-to-multipoint communication relies on the IP network infrastructure to deliver
361 messages to one or more Recipients. IP network devices – like network routers, switches, or bridges – are
362 used for this purpose. A transport layer multicast messaging pattern is much more efficient than sending
363 multiple point-to-point transport layer unicast messages. Multicast allows near-simultaneous message
364 delivery and, thereby, near-simultaneous modification of actuators on all devices in a group, providing
365 users with a perceived effect of synchronism or simultaneity (lights ON). Multicast is only possible on
366 top of UDP in IP networks while using a sequence number supports recognizing duplication of messages.
367 However, IP multicasting is unreliable (unconfirmed transmission) and requires proper configuration of
368 suitable network equipment like (managed) switches and routers. Routers need knowledge about listeners
369 in their subsystem for efficient message filtering. Multiple transport layer unicast MAY be used instead of
370 multicast for communication where simultaneity is of lower importance and/or reliability is of higher
371 priority.

The advantages of a point-to-multipoint system include the following:

- Near-simultaneous message delivery to multiple Recipients with UDP protocol using multicast addressing.
- Only standard network infrastructure is needed.
- Messages are delivered directly without software intermediaries and, therefore, have less latency and overhead.

The Group Notification Message Keys used in the below figure are defined in clause 2.5.9.

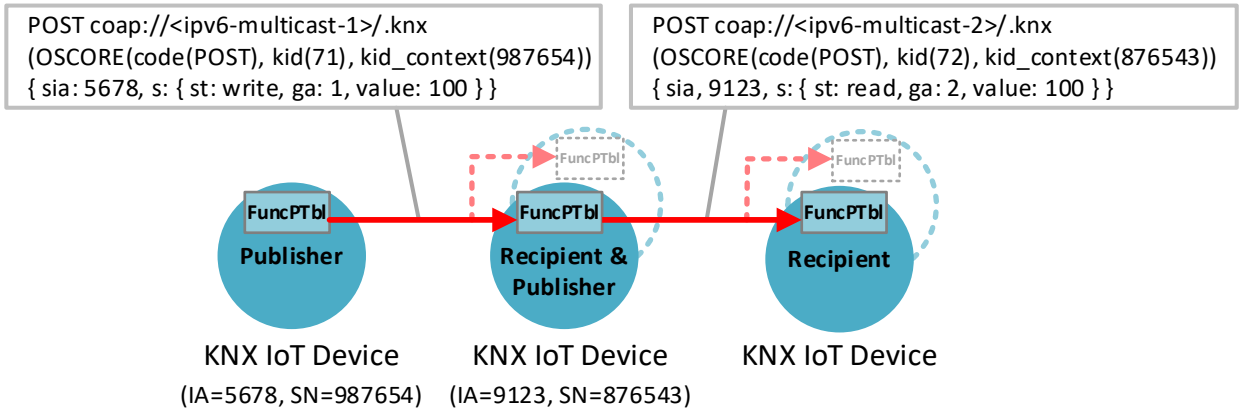


Figure 5 - Brokerless system [i)]

Figure 5 shows a typical brokerless communication scenario with two different group connections, where a sender (Publisher) uses multicast addressing to reach one or multiple Recipients. The device shown in the middle acts as Recipient for the 1st group and as the sender (Publisher) for another 2nd group.

The multicast IPv6 address format for point-to-multipoint communication is defined in clause 2.6.5, "Multicast Group IP Addresses".

The unicast IPv6 address format for point-to-point communication is defined in clause 2.6.2, "Device IP Address".

Application-level security, based on OSCORE (see clause 3.6, "OSCORE Application Layer Security"), ensures message confidentiality and integrity for end-to-end security between Publisher and Recipient.

2.3.3 Message Broker-based System

Point-to-point messaging with a fixed (configured) setup or Broker-based Publish–Subscribe (pub/sub) setup is a messaging pattern for facilitating communication in loosely coupled distributed systems. It has gained increasing popularity in the context of Internet of Things (IoT) applications.

With a fixed point-to-point configuration, the serving device sends the message directly to the Recipient using unicast transport.

In most pub/sub systems, a reliable transport layer unicast communication is used. Hence, these systems rely on unicast application layer messaging (i.e., point-to-point connections from a central broker to each Subscriber) to disseminate messages. This option assumes a Message Broker in the middle. All the communication is passed through the Message Broker, and the Message Broker also routes the group notification to all subscribed applications based on a mapping table.

The advantages of this model include the following:

- Publisher and Subscriber do not have to be directly addressable. They can be anywhere if they have access to the Message Broker.
- Publisher and Subscriber can use different messaging protocols to communicate with the Message Broker (e.g., KNX Classic, WebSocket, HTTP, or CoAP).
- Reliable Group Notifications (acknowledged unicast messages) can be handled to a large list of Subscribers, multiple networks, or even chained Message Brokers or scalable Message Brokers.

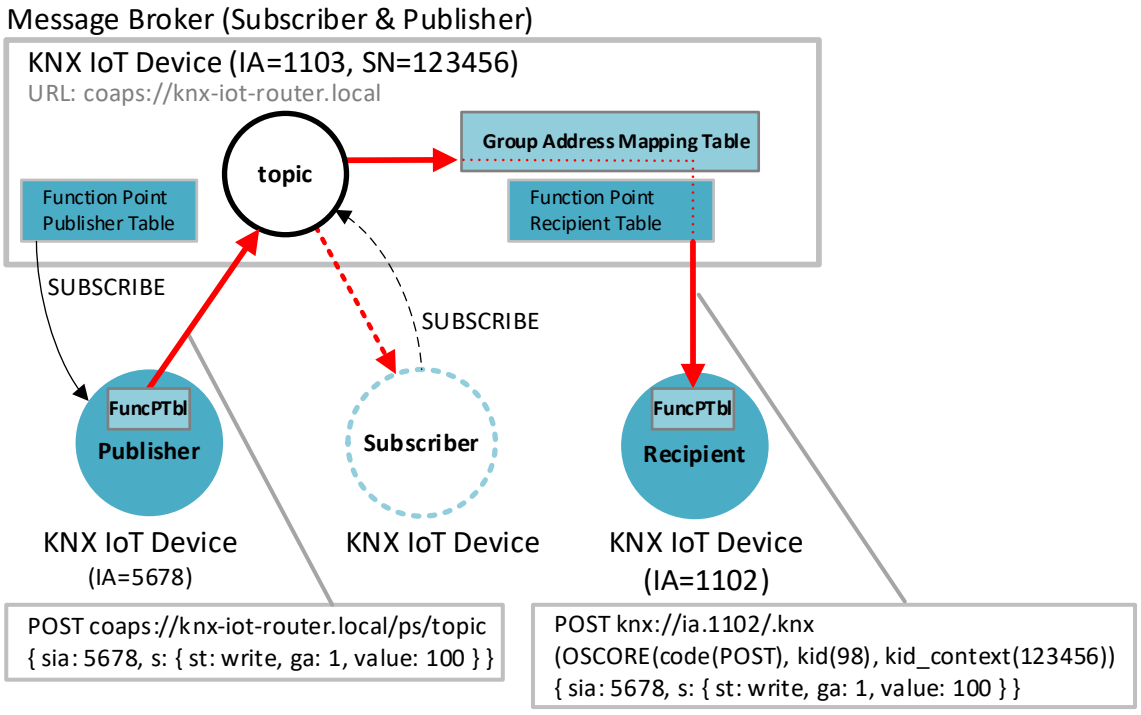


Figure 6 – Broker-based System [i)]

Figure 6 shows a typical communication scenario with a Message Broker, a (D)TLS Publisher, and an OSCORE Recipient. Note that the "kid" in the OSCORE message always contains the OSCORE input material identifier (osc-id) from the corresponding access token, and the "kid_context" contains the configured context in the OSCORE access token from the device that sends the message. It requires a Message Broker using technologies like AMQP, MQTT, etc., to bridge between different transport layers and protocols. In this example, the Message Broker relays messages from (D)TLS Publishers to Recipients (or Subscribers). A Message Broker can also queue and send the same message to multiple Recipients. Note that the Message Broker functionality and the protocol proxy (e.g., from CoAP to MQTT) are outside the scope of this specification. In the context of KNX IoT, the Message Broker is just a CoAP server and an intermediate that delivers messages to KNX IoT Recipients. Hence, the Message Broker is a standard KNX IoT device, and KNX IoT defines how to connect to a Message Broker in a reliable and secure manner.

The unicast IPv6 address format is defined in clause 2.6.2, "Device IP Address".

Message confidentiality and integrity with the Message Broker can be ensured at two levels.

1. Application-level security as end-to-end security between Publisher and Recipient based on OSCORE, see clause 3.6 "OSCORE Application Layer Security" and/or
2. Transport security between Publishers/Subscribers and the Broker with (D)TLS.

The application-level security requires all Publishers, Subscribers, and Recipients to have pre-shared credentials that grant them access to the necessary group. The configuration of credentials and access scopes is described in clause 3.5, "Device Access Control".

2.3.4 Device Linking

2.3.4.1 Introduction

KNX IoT devices send and receive group messages on configured unicast or multicast IP addresses (messaging endpoint). A MaC (Management Client) configures the KNX system communication interworking. For this purpose, the MaC configures the KNX Function Point Tables of all KNX devices in an installation. A KNX IoT device SHALL support Group Communication via

- Transport layer unicast and multicast (see clause 2.3.2,"Brokerless System"),
- Publish-Subscribe (see clause 2.3.3, "Message Broker-based System")

2.3.4.2 Function Point Tables

The Function Point Tables of a device are *Group Address* mapping tables. The Function Point Tables have references from *Group Addresses* ("ga") of a KNX IoT device to a protocol-specific (e.g., CoAP) destination address for sending or source address for receiving messages (unicast or multicast URL).

The MaC configures Function Point Tables on all KNX IoT Point Devices in both a brokerless and a broker-based system. The Function Point Tables consist of the following tables:

- Point mapping configuration: The Group Object Table maps *Group Addresses* to a Point to read a value, store a value, or further processing of incoming values (see clause 2.5.7.3, "Group Object Table Resource (fp/g)").
- Outgoing group event configuration: The Function Point Recipient Table (destination unicast/multicast address) defines to which devices or with which multicast address group events are sent (see clause 2.5.7.4, "Function Point Recipient Table Resource (fp/r)").
- Incoming group event configuration: The Function Point Publisher Table (unicast subscription or multicast listening address) contains configurations from which devices group events need to be received (subscription address) or from which multicast address group events are received (see clause 2.5.7.5, "Function Point Publisher Table Resource (fp/p)").

The Group Object Table of a device contains all *Group Object* Points ("href": {point-path}) that can be activated; the *Group Object* Points are defined by the manufacturer.

The KNX IoT *Group Address* is used for KNX S-Mode communication. The *Group Address* is an unstructured ID ("ga"). The MaC assigns the "ga" with predefined *Group Objects* (Group Object Table) and writes the configuration to the KNX IoT device (see clause 2.4, "Device Bootstrapping and Configuration"). In addition, the *Group Address* is also used to discover configured S-Mode *Group Objects* (see clause 2.6.1, "Discovery").

The following figure illustrates the general linkage concept between *Group Objects*, Points, and *Group Addresses* but does not prescribe how to implement the table structures.

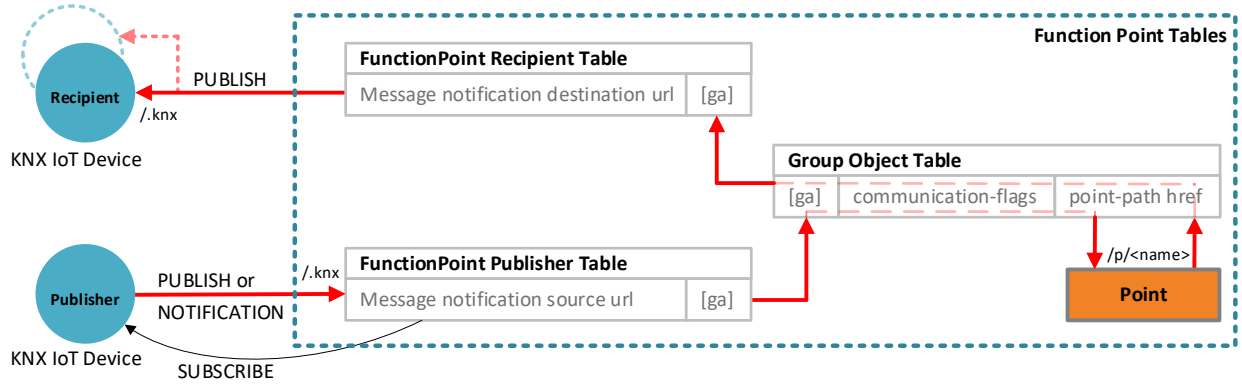


Figure 7 – Function Point Tables

2.4 Device Bootstrapping and Configuration

2.4.1 Introduction

Device Bootstrapping is the process of joining a device to an installation and configuring an application. The application configuration of a KNX IoT Point API device aligns with the existing MaC (ETS) workflows and contains the following steps.

STEP 0 - Project creation in MaC.

This is as it is done in KNX Classic. This can be done before, in parallel, or after step 1. This may include at this place as well offline configuration, i.e., setting parameters and creating links between the devices for installation parts engineered entirely offline.

STEP 1 - Enter KNX IoT Point API device *KNX Serial Number* in MaC, e.g., by QR code scan.

This step brings the serial number to MaC, needed for step 2, and, e.g., an authentication code required for step 3.

STEP 2 - Device Discovery.

Discovery of a KNX IoT Point API device follows the method described in 2.6.1.2, "Device Discovery with DNS-SD".

NOTE 1 With this version of the KNX IoT Point API specification, it is assumed that a device is reachable already on the IP network at this step, e.g., a wireless (Thread or Wi-Fi) KNX IoT device is already onboarded into the IP network and thus discoverable.

STEP 3 - Device Assignment, i.e., matching the discovered devices with the instances foreseen in the MaC project.

This step includes the security setup described under clause 3.2, "Device Identity Enrollment", and 3.5, "Device Access Control".

Device Individualization, i.e., assigning a *KNX Individual Address*, is the same as in KNX Classic. This is either by using the *Programming Mode* Method or the *KNX Serial Number* Method in case the *KNX Serial Number* is already known to MaC, e.g., by QR code scan.

Device Individualization is described in more detail in clause 2.4.2, "Device Individualization Procedure".

STEP 4 - Offline configuration.

This step configures parameters and creates links between the devices (Function Point Table). At this place, this may also include configuration of a before online discovered and uploaded installation or part of an installation.

STEP 5 - Download device configuration (Application Download with MaC).

With this step, MaC configures the credentials at application level, pre-shared keys. This is, once downloaded, the devices can trust each other.

Device configuration is described in more detail in clause 2.4.3, "Device Configuration Procedure".

The above numbering does not imply a fixed order for executing these steps, and the order may differ as far as possible and meaningful. Some may even be repeated several times during setup/commissioning of an installation, as it is done in KNX Classic.

NOTE 2 This version of the KNX IoT Point API specification does not consider how to join, replace or delete a device from the network and how to avoid the replaced/removed device can still communicate.

2.4.2 Device Individualization Procedure

Every device in a KNX installation SHALL have a unique, unambiguous *KNX Individual Address*, and this clause defines how a MaC configures the KNX IoT device's *KNX Individual Address*.

The *KNX Individual Address* configuration workflow is like the existing procedure in KNX Classic. The device can either be identified by its *KNX Serial Number* or by activating the KNX *Programming Mode* locally on the device, for example, by pressing a button. The MaC can choose one of the following discovery options:

- DNS-SD: see clause 2.6.1.2 "Device Discovery with DNS-SD" with _knx service type.
- "/.well-known/core": see clause 2.6.1.3 "Resource Discovery with CoAP".
  - with *KNX Serial Number*, see clause 2.6.1.3.6.1 "Endpoint Name "ep"".
  - with *Programming Mode*, see clause 2.6.1.3.6.2 "Programming Mode "if.pm"".

Before writing the *KNX Individual Address* to the discovered device, a MaC SHALL check that no other device has the same *KNX Individual Address*. The MaC has the following options to check *KNX Individual Addresses*:

- DNS-SD: see clause 2.6.1.2 "Device Discovery with DNS-SD" with _ia subtype.
- "/.well-known/core": see clause 2.6.1.3 "Resource Discovery with CoAP" or the example below.

Verify whether the *KNX Individual Address* is already occupied on the network.

REQ:
GET coap://[FF03::FD]/.well-known/core?ep=knx://ia.33a3.20a

RES:
2.05 CONTENT (Content-Format: application/link-format (40))
Payload:
<>;ep="knx://sn.1caffe1234 knx://ia.33a3.20a"

If the MaC receives a response to a multicast request, it SHALL conclude that the *KNX Individual Address* is occupied. If no response is received after a timeout, then the *KNX Individual Address* is not occupied, and subsequently, the MaC can configure the *KNX Individual Address*. For this configuration request, the MaC uses the sender IP address out of the IP response to the before-discovery request as the destination device IP unicast address.

The procedure SHALL deactivate the *Programming Mode* by restarting the device ("/.well-known/knx").

At this point, the MaC knows the IP address of the uncommissioned KNX IoT device. However, the MaC MAY configure the security credentials first before it configures the *KNX Individual Address* (see clause 3.6.2)

In the last step, the MaC configures the device's *KNX Individual Address* (see clause 2.5.5.5). A KNX IoT device SHALL accept this configuration request and set its *KNX Individual Address* accordingly.

Write device *KNX Individual Address.*

REQ:
POST coap://{ipv6-unicast}/.well-known/knx/ia
(Content-Format: application/cbor (60), OSCORE(code(POST), kid(osc-id)))
Payload:
{ 12: <*KNX Individual Address*>, 25: <fid>, 26: <iid> }

534 ### 2.4.3 Device Configuration Procedure

535 #### 2.4.3.1 General

536 After the device individualization procedure, the MaC writes the device's new Function Point
537 configuration (fp/g, fp/r, fp/p) and application configuration parameters (p) to the device. A Device Load
538 State Machine controls these resources (see clause 2.5.8, "Application Program Object Resource (ap)").

539 #### 2.4.3.2 Full download

540 Table 3 specifies the configuration procedure for a full download. An example is given in clause 6.2.1,
541 "Full Download Example".

542 **Table 3 – Full download**

Step	Procedure	Description
1	Check MaC DB Entry compatibility with KNX IoT device by comparing "dev/mid" and "dev/hwt".	See clause 2.5.6, "Device Object Resource (dev)". Continue with the next steps only if the compatibility check is successful.
2	Set State Machine to "unload".	See clause 2.5.8, "Application Program Object Resource (ap)". If the intermediate state "unloading" is supported by the KNX IoT device (Management Server), it may answer with state "unloading". The MaC SHALL NOT continue before the KNX IoT device has reached the state "unloaded".
3	Set State Machine to "loading".	See clause 2.5.8, "Application Program Object Resource (ap)".
4	Write parameter configuration.	See clause 2.5.11, "Parameter and Diagnostic Property Resource (p)".
5	Write Group Object Table.	See clause 2.5.7.3, "Group Object Table Resource (fp/g)".
6	Write Function Point Publisher Table.	See clause 2.5.7.5, "Function Point Publisher Table Resource (fp/p)".
7	Write Function Point Recipient Table.	See clause 2.5.7.4, "Function Point Recipient Table Resource (fp/r)".
8	Write Application Program Version "ap/pv".	See clause 2.5.8, "Application Program Object Resource (ap)".
9	Set State Machine to "loaded".	See clause 2.5.8, "Application Program Object Resource (ap)". If the intermediate state "loadcompleting" is supported by the KNX IoT device, then it MAY answer with the state "loadcompleting". The MaC SHALL NOT continue before the KNX IoT device has reached the state "loadcomplete".
10	Read the device configuration fingerprint of the written resources.	See clause 2.5.5.4, "Device Configuration Fingerprint (.well-known/knx/f)". The retrieved checksum MAY be stored by the MaC for a future decision about a partial/differential download.
11	Restart device (optional).	See clause 2.5.5.3.3, "(Basic) Restart Command".

543 #### 2.4.3.3 Partial/Differential download

544 Table 4 specifies the configuration procedure for a partial/differential download. The partial/differential
545 download is used if the device is in a known state. In this case, the MaC can write only the differences to
546 the device. An example is given in clause 6.2.2, "Partial Download Example".

547 **Table 4 – Partial/differential download**

Step	Procedure	Description
1	Check MaC DB Entry compatibility with KNX IoT device by comparing “dev/mid” and "dev/hwt".	See clause 2.5.6, "Device Object Resource (dev)". Continue with the next steps only if the compatibility check is successful.
2	Check if the KNX IoT device is in a known state.	See clause 2.5.5.4 "Device Configuration Fingerprint (.well-known/knx/f)". If the device configuration fingerprint value is known by the MaC, then the MaC MAY continue with a differential download. Else, the MaC SHALL fall back to the full download procedure acc. Clause 2.4.3.2, "Full download", and continue there with Step 2.
3	Set State Machine to "loading".	See clause 2.5.8, "Application Program Object Resource (ap)".
4	Write parameter configuration (conditional: only if changes there).	See clause 2.5.11, "Parameter and Diagnostic Property Resource (p)". The MaC MAY write only changed parameters (identified by href).
5	Write Group Object Table (conditional: only if changes there).	See clause 2.5.7.3 "Group Object Table Resource (fp/g)". The MaC MAY write only changed group objects (identified by "id").
6	Write Function Point Publisher Table (conditional: only if changes there).	See clause 2.5.7.5, "Function Point Publisher Table Resource (fp/p)". The MaC MAY write only changed table entries (identified by "id"). Entries with a new "id" will create a new entry.
7	Write Function Point Recipient Table (conditional: only if changes there).	See clause 2.5.7.4, "Function Point Recipient Table Resource (fp/r)". The MaC MAY write only changed table entries (identified by "id"). Entries with a new "id" will create a new entry.
8	Write Application Program Version “ap/pv”	See clause 2.5.8, "Application Program Object Resource (ap)".
9	Set State Machine to "loaded".	See clause 2.5.8, "Application Program Object Resource (ap)". If the intermediate state "loadcompleting" is supported by the KNX IoT device, then it may answer with the state "loadcompleting". The MaC SHALL NOT continue before the KNX IoT device has reached the state "loadcomplete".
10	Read the checksum of the written resources.	See clause 2.5.5.4, "Device Configuration Fingerprint (.well-known/knx/f)". The retrieved checksum MAY be stored by the MaC for a future decision about a partial/differential download.
11	Restart device (optional).	See clause 2.5.5.3.3, "(Basic) Restart Command".

2.5 Resource Model

2.5.1 Introduction

Resources are the basis for representing the state and functionality or, rather, a KNX application. Resources provide access to standard application services like device and resource discovery, device-specific properties, and function calls. The resource model provides means for presenting, describing, and addressing data and metadata. Data is accessible by RESTful methods for creating, reading, setting, and deleting data. This clause gives rules for device interactions that need to be exposed for configuration and discovery and provides guidelines for using predefined attributes.

2.5.2 Resources (Points)

2.5.2.1 Definition

Points are KNX application-specific API resources representing the KNX application domain model. With a URI, a resource is uniquely defined, and a URL includes the schema to access the resource. Using KNX-specific nomenclature for resource names helps developers understand the functionality and basic semantics of resources, and it also reduces the need for further documentation of the API items. This clause defines mandatory and optional resource paths and names.

2.5.2.2 Base Path

A KNX IoT device MAY have a base path to which the Point API resource paths are appended. However, it is RECOMMENDED to avoid a base path if possible.

2.5.2.3 Resource Names

To simplify the encoding of resource IDs in URLs, their representations SHALL only consist of ASCII strings of letters, numbers, underscore, minus, colon, and period.

URLs are often treated as case sensitive; therefore, lowercase separated words with hyphens "-" SHOULD be used.

This specification does not define resource names for *Functional Blocks* (f) or properties (p). However, it is RECOMMENDED to reuse either IDs or names from the KNX Application Descriptions (KNX standard Volume 7):

- Functional Block path: "/{base-path}/f/{fb-id-instance}"
- Property path: "/{base-path}/p/{fb-id-instance}/{property-id}"

2.5.2.4 Resource Path

The resource path definitions allow access to items of a KNX IoT device, such as discovery, device, authorization, *Functional Blocks*, and Property items, based on entities as specified in [03] "KNX Information Model".

To keep the API and authorization simple, a resource path SHOULD contain only one collection or item resource with an {id} and MAY a sub-resource path for collections:

- collection path: "/collection"
- item path: "/items/{id}"
- collection path of a particular item: "/items/{id}/collection"

It is vital to keep maintenance and service evolution manageable. Therefore, the resources follow "functional segmentation" and "separation of concern" design principles and do not mix different functionalities in the same API resource. This keeps maintenance and service evolution manageable.

The length of a KNX IoT device resource path SHALL not exceed the maximum of 30 bytes.

589 A KNX IoT device server has the following MANDATORY ("M") or OPTIONAL ("O") resource path
590 definitions with their corresponding access methods.

591 **Table 5 – Resource path definitions**

Category	Method	Resource Path	Support	Reference
Discovery	GET	/.well-known/core	M	clause 2.5.4
Device	POST	/.well-known/knx	M	clause 2.5.5
	POST	/.well-known/knx/ia	M	clause 2.5.5.5
	GET	/.well-known/knx/f	M	2.5.5.4
	GET	/{base-path}/dev	M	clause 2.5.6
	GET	/{base-path}/dev/{property-path}	M	
	PUT	/{base-path}/dev/{property-path}	M/O	
	GET	/{base-path}/swu	M	clause 4.2
	GET	/{base-path}/swu/{property-path}	M	
	PUT	/{base-path}/swu/{property-path}	M/O	
Application Program	GET	/{base-path}/ap	M	clause 2.5.8
	GET	/{base-path}/ap/{property-path}	M	
	PUT	/{base-path}/ap/{property-path}	M/O	
Security	POST	/.well-known/knx/spake	M	clause 3.6.3.3
	PUT	/.well-known/knx/spake	M	clause 3.6.3.5
	GET	/.well-known/knx/idevid	O	clause 3.3.1
	GET	/.well-known/knx/ldevid	O	clause 3.3.2
	GET	/{base-path}/auth	M	clause 3.2.5
	POST	/{base-path}/auth/crts	O	clause 3.5.2
	GET	/{base-path}/auth/crts	O	
	DELETE	/{base-path}/auth/crts/{cert-id}	O	
	POST	/{base-path}/auth/at	M	clause 3.5.4
	GET	/{base-path}/auth/at	M	
	GET	/{base-path}/auth/at/{token-id}	M	
	DELETE	/{base-path}/auth/at/{token-id}	M	
	GET	/{base-path}/auth/o/{property-path}	M	clause 3.6.4
Messaging	POST	/.knx	M	clause 2.5.9
	GET	/.knx	M	
Function Point Tables	POST	/{base-path}/fp/g	M	clause 2.5.7
	GET	/{base-path}/fp/g	M	
	GET	/{base-path}/fp/g/{group-object-id}	M	
	DELETE	/{base-path}/fp/g/{group-object-id}	M	
	POST	/{base-path}/fp/r	M	
	GET	/{base-path}/fp/r	M	
	GET	/{base-path}/fp/r/{recipient-id}	M	
	DELETE	/{base-path}/fp/r/{recipient-id}	M	
	POST	/{base-path}/fp/p	O	

Category	Method	Resource Path	Support	Reference
	GET	/{base-path}/fp/p	O	
	GET	/{base-path}/fp/p/{publisher-id}	O	
	DELETE	/{base-path}/fp/p/{publisher-id}	O	
Actions	POST	/{base-path}/a/lsm	M	clause 2.5.8
	GET	/{base-path}/a/lsm	M	
	POST	/{base-path}/a/swu	O	clause 4.2
	POST	/{base-path}/a/sen	O	clause 3.2.4.2
Functional blocks	GET	/{base-path}/f/{fb-id-instance}	M	clause 2.5.10
Properties	POST	/{base-path}/p	M	clause 2.5.11
	GET	/{base-path}/p	M	
	GET	/{base-path}/p/{property-path}	M	
	PUT	/{base-path}/p/{property-path}	M	
Subscriptions	DELETE	/{base-path}/sub	M	clause 2.5.12

2.5.3 Interface Types (if)

2.5.3.1 Introduction and overview

This clause specifies a set of default interface types and defines how manufacturers can add additional manufacturer-specific interface types.

The interface describes a generic context or view to interact with a resource or a set of resources. The interface description "if" attribute is an opaque string that provides a name indicating a specific interface definition used to interact with the KNX IoT device. The usage of the interface description "if" attribute is described in 3.5.3 "Access Scope".

The interface types are derived from the KNX Device Model (see clause 1.3), and MAY contains a set of additional mandatory or optional metadata.

Table 6 defines KNX IoT interfaces and which CoAP Methods are allowed for a particular interface on a KNX IoT device:

Table 6 – Interface definitions and methods

Type	Interface	Method	Description
Link List	if.ll	GET, (OBSERVE)	Read a linked list and, in combination with if.o, subscribe to all Points of the list.
Parameter	if.p	GET, PUT, (OBSERVE)	Adjust parameter Point (see [03] "KNX Information Model").
Diagnostic	if.d	GET, (OBSERVE)	Read diagnostic Point (see [03] "KNX Information Model").
Configuration	if.c	GET, PUT, POST, DELETE	Configuration and programming of a device.
Logical Input	if.i	PUT, POST	Write and command runtime input Group Object Point (see [03] "KNX Information Model" and clause 1.3).
Logical Output	if.o	GET, POST, OBSERVE	Read and subscribe runtime output Group Object Point (see [03] "KNX Information Model" and clause 1.3).

Type	Interface	Method	Description
Group Communication	if.g.s	POST, GET (OBSERVE)	Group communication (S-Mode) runtime interworking (input and output) address.
Batch	if.b	GET, PUT, POST	Read or write a collection (e.g., Point list).
Actuator	if.a	GET, PUT, POST	Hardwired actuator (see [03] "KNX Information Model" and clause 1.3).
Sensor	if.s	GET, PUT	Hardwired sensor (see [03] "KNX Information Model" and clause 1.3).
Security	if.sec	GET, PUT, POST, DELETE	Configuration (read and write) of security, incl. authorization-related data (see clause 3 "Security").
Software Update	if.swu	GET, PUT, POST, DELETE	Software update (push and pull) related data (see clause 4).
Programming Mode	if.pm	GET	Data that can be read in Programming Mode.
Manufacturer	if.m.{name}	Manufacturer-specific definition	Manufacturer-specific interface types.

2.5.3.2 Link List Interface Type (if.ll)

The Link List interface gradually reveals resources on a KNX IoT device. It is used to retrieve (GET) a list of resources on a KNX IoT device. The GET request SHOULD contain an Accept option with the application/link-format content format. The request returns a list of URI references with absolute paths to the resources as defined in CoRE Link Format [RFC6690]. This interface is typically used with a parent resource to enumerate sub-resources but MAY be used to reference any resource on an origin server.

CoAP client requests a link list (datapoint list) from a Room Temperature Sensor (Functional Block 321).

REQ:

```
GET coap://{ip unicast}/f/rts
```

RES:

CONTENT (Content-Format: application/link-format (40))

Payload:

```
</p/rts/temproom>;rt=":dpa.321.51";ct=50,
</p/rts/tempcorrvalue>;rt=":dpa.321.111";ct=50,
</p/rts/tempalarmlimitupper>;rt=":dpa.321.113";ct=50
```

2.5.3.3 Parameter and Diagnostic Interface Type (if.p, if.d)

Parameter and diagnostic Properties (see clause 1.3) are the basic interfaces of a Point API and define whether a client can only read or read and update values and metadata, such as getting the current sensor value or updating a setpoint. The following MANDATORY and OPTIONAL metadata members from clause 2.5.11.3 are assigned to the "if.p" (parameter) and "if.d" (diagnostic) interface:

- id
- rt
- if
- unit
- min
- max

A Point MAY implement a Parameter ("if.p") and Diagnostic Property ("if.d") in combination with a communication interworking interface type ("if.i", "if.o"). In such cases, it is RECOMMENDED to combine only "if.i" with "if.p" (inputs) or "if.o" with "if.d" (outputs).

2.5.3.4 Configuration Interface Type (if.c)

The Configuration interface is used to configure a basic functionality, Function Point Tables, or for programming a KNX IoT device.

2.5.3.5 Input Interface Type (if.i)

A communication interworking logical input Point (see clause 1.3) receives events from Publishers. An input interface Point receives either status events (e.g., light switch ON/OFFF) or periodic updates from Publishers (if.o interface). A Point with an interface type "if.i" SHALL NOT be of interface type "if.o" at once.

The Recipient device MAY have a default behavior if no update is received within a given period (e.g., Max-Age timeout). For example, a controller gets a periodic synchronization from an output temperature sensor and uses a default value if the sensor value is not present (e.g., broken or not configured). It is RECOMMENDED to use the "hbt" metadata member for timeout configurations in combination with the following timeout calculation: timeout = ("hbt" * 2) + 1.

Example: timeout = (15min*2) + 1 = 31min

2.5.3.6 Output Interface Type (if.o)

A communication interworking logical output Point (see clause 1.3) sends status events (e.g., light switch ON/OFFF) or periodic updates (e.g., temperature sensor) from a Publisher to a Receiver ("if.i" interface). A Point with an interface type "if.o" SHALL NOT be of interface type "if.i" at once.

An output interface Point either supports subscriptions (e.g., CoAP Observe) or S-Mode event notifications. Examples are operating modes, setpoints, status updates, sensor values, etc. Those values, such as an outside temperature value, MAY have additional metadata used to synchronize values throughout the system. The following metadata members from clause 2.5.11.3, "Metadata Query Parameter "m", MAY be assigned to the "if.o" interface:

- mrt
- cov
- hbt

2.5.3.7 Group Object Interface (if.g.s)

Points with a Groups Object interface support S-Mode event notifications, and a MaC can link *Group Addresses* to this object. The following metadata members from clause 2.5.11.3, "Metadata Query Parameter "m", are assigned to the "if.g.s" interface:

- g

2.5.3.8 Batch Interface Type (if.b)

The Batch interface ("if.b") manipulates a collection of sub-resources or an array of values with a single request. A resource with an "if.b" interface SHALL support pagination as defined in clause 2.6.8, "Pagination". Single items can be accessed with pagination query parameters starting with pn=0 if no query parameter is present.

If a resource contains an array of datapoint values, then the resource SHALL be of type Batch interface ("if.b").

Datapoint values SHALL be modeled as key/value pairs in the GET response (see rules in 2.5.11.1), but only if the client requests more than one item (ps > 1). The default list size SHALL be ps=1 if the query parameter is absent.

The KNX IoT device GET response SHALL list items in the same order as they were written (first array item = "po=0"). A KNX IoT device SHALL support write (PUT) of a single value, for example, on "if.p" interfaces. However, updating a list of values on the same resource is OPTIONAL.

The CoAP client reads an array of Ipv6 configurations with a single request.

REQ:

```
GET coap://{ipv6-unicast}/dev/ipv6?pn=0&ps=3
```

RES:

CONTENT (Content-Format: application/cbor (60))

Payload:

```
[
    { 1: "20010db8000300006cd98ad28e881a47" },      //value
    { 1: "fe800000000000006cd98ad28e881a47" },      //value
    { 1: "fd00000ff1ce0ba56cd98ad28e881a47" }       //value
]
```

The client omits pagination parameters and gets only the first list item.

REQ:

```
GET coap://{ipv6-unicast}/dev/ipv6
```

RES:

(Content-Format: application/cbor (60))

Payload:

```
{ 1: 20010db8000300006cd98ad28e881a47 }     //value
```

The interface type of a resource representing a collection of sub-resources SHALL be of type Batch interface. The Batch interface resource SHALL support the same methods as its sub-resources and can be used to read (GET), update (PUT), or apply (POST) the values of that sub-resource. The method used on Batch only applies to sub-resources that support it. For example, sensor interfaces do not support PUT; thus, a PUT request to a Sensor member of that Batch would be ignored.

The next example shows how a MaC configures multiple values of one or many Properties with a single request.

CoAP client writes a collection of datapoints to a device with a single request to a Room Temperature Sensor (Functional Block 321).

REQ:

POST coap://{ipv6-unicast}/p (Content-Format: application/cbor (60))

Payload:

```
[
    { 11: "/p/rts/tempcorrvalue", 1: 0.5 },
    { 11: "/p/rts/tempalarmlimitupper", 1: 27.2 }
]
```

2.5.3.9 Programming Mode Interface Type (if.pm)

A MaC MAY want to find devices in *Programming Mode* during the device bootstrapping and configuration step. Hence, a KNX IoT device SHALL support the *Programming Mode*.

With the *Programming Mode*, the destination KNX IoT device can be identified by selecting the device manually. This can be done by pressing a button that brings this device into *Programming Mode*, i.e., only the device where the button is pressed. The way a KNX IoT device is set to *Programming Mode* is manufacturer specific.

2.5.3.10 Manufacturer Specific Interface Type (if.m.{name})

The manufacturer interface allows manufacturer-specific interface types on resources, such as manufacturer-specific methods (GET, PUT, etc.). Manufacturer-specific interface types SHALL start either with "urn:" or ":if.m." followed by a manufacturer-specific string {name}.

2.5.4 Device Discovery Resource (.well-known/core)

For CoAP-based resource discovery purposes, a KNX IoT device SHALL support the /.well-known/core resource [RFC6690], and the device SHALL accept unicast and multicast discovery requests on this resource. More details are described in clause 2.6.1, "Discovery". In the case of multicast discovery queries, a KNX IoT device SHALL follow the rules for response suppression described in clause 2.6.4.3, "Response suppression". In the case of unicast discovery queries, a KNX IoT device SHOULD respond as described in clause 2.6.3.3.2, "Error Responses to Queries".

The response SHALL contain all *Functional Block* resources of a KNX IoT device. The response MAY have only *Functional Block* resources to reduce the message size. However, the response SHALL contain all implemented *Functional Blocks* standardized in KNX Application Descriptions (KNX standard Volume 7) and supported content types (ct). In addition, the following resources SHALL be in the response if supported by the device:

- /{base-path}/dev
- /{base-path}/auth
- /{base-path}/swu
- /.knx

The *Functional Block* resources in the response SHALL support the application/link-format. This allows clients to discover Points with subsequent requests (see clause 2.5.10, "Functional Block Resource (f)").

The response SHALL contain the namespace identifier "urn:knx" since the device context is unknown to the client at this stage.

CoAP client discovers device Functional Blocks.
REQ: GET coap://{ipv6-unicast}/.well-known/core **RES:** CONTENT (Content-Format: application/link-format (40)) **Payload:** </auth>;rt="urn:knx:fb.auth";ct=40, </dev>;rt="urn:knx:fb.0";ct=40, </swu>;rt="urn:knx:fb.swu";ct=40, </.knx>;rt="urn:knx:g.s"ct=60, </f/rts>;rt="urn:knx:fb.321";ct=40

2.5.5 Device API Resource (.well-known/knx)

2.5.5.1 General requirement

A KNX IoT device SHALL support the "/.well-known/knx" resource for KNX-specific device operations, such as device individualization, security configuration (see clause 3, "Security"), reset to the default state, or device restart.

2.5.5.2 API Version Resource Object

The "/.well-known/knx" resource also provides the KNX IoT API version information and relationship information to identify the API entry point for a MaC or any other client. A basic response contains the data listed in Table 7. MANDATORY ("M") attributes SHALL be present in the message.

Table 7 – Data in the basic response

JSON Key	CBOR Key	CBOR Type	Support	Description
"api"	"api"	map	M	The API object contains the most recent supported API version on the KNX IoT device.
"version"	"version"	Text string	M	Major and minor version of the KNX IoT Point API ("<major.minor.patch>").
"base"	"base"	Text string	M	The /{base-path}/ member value is part of the entire URL of a KNX IoT Point API. The device SHALL reply with "/" if the {base-path} is empty or at the end of the {base-path}. This simplifies concatenation with relative paths. A client SHALL include the resource path base member value when requesting content from a KNX IoT device (see clause 2.5.2.2, "Base Path").

Semantic versioning contains major, minor, and patch information. The version member value SHALL contain the major, minor, and patch number according to the format major.minor.patch, represented as a string. The major number starts from the value 1; the minor and patch numbers begin with the value 0.

On any compatible change with the current KNX IoT Point API, the major number remains, and the minor number SHALL be incremented. The base member value SHALL NOT be changed.

On any incompatible change with the current KNX IoT Point API, the new major number SHALL be incremented. The new minor number SHALL be set to 0.

The KNX IoT device Point API SHALL have version 1.0.0 if the API complies with this specification version and implements the MANDATORY resource definitions defined in clause 2.5.2.4 "Resource Path".

Table 8 defines MANDATORY ("M") resources of a KNX IoT device and the corresponding resource path names that SHALL be used.

Table 8 – Mandatory and option configuration resources

Resource path	rt & Data Types	Format	Method	Support	Request/Response	Notes

/.well-known/knx	NA	cbor (json)	GET	**M**	Res: Content-Format: application/json
					Payload:
					{
					"api": {
					"version": "1.0.0",
					"base": "/"
					}
					}
						Device KNX Point API version and base path.

2.5.5.3 Device Command Resource Object

2.5.5.3.1 Overview

The functionality of the reset resource is derived from the functionality of the Master Reset Service as defined for KNX Classic devices, see [09].

A KNX IoT device SHALL support the reset command (cmd), with supported erase codes 0x2 ("Reset to default state") and 0x7 ("Reset to default without IA").

With erase code 0x2 ("Reset to default state"), all addressing information and security configuration data SHALL be reset to the default state. With erase code 0x7 ("Reset to default without IA"), all configuration data SHALL be reset to default state except addressing information (IA, Device IP address, Installation ID) and security configuration data (credentials) that are needed after the reset to access the device without the need to discover the device again and/or renew addressing information and security credentials.

Table 9 defines JSON keys and the CBOR mapping for CoAP request and response members of "/.well-known/knx". MANDATORY ("M") attributes SHALL be present in the message, and OPTIONAL ("O") MAY are present in the message, but all attributes SHALL be supported on the device.

754 **Table 9 – Command members of "/.well-known/knx"**

JSON Key	CBOR Key	CBOR Type	Support	Description
"value"	1	unsigned	**M**	Erase-code value (see [09], Table 4).
"cmd"	2	text string	**M**	Service command. Enum: reset, restart.

755

756 **Table 10 – Response members of "/.well-known/knx"**

JSON Key	CBOR Key	CBOR Type	Support	Description
"code"	"code"	unsigned	**M**	Device response error code (see [09], Table 5).
"time"	"time"	unsigned	**M**	Process time (see [09]).

757

758 2.5.5.3.2 (Master) Reset Command

759 A KNX IoT device provides the functionality to reset it to its default state, which typically corresponds to
760 the ex-factory state of the device's configurable data. Resetting a device to the default state is a device-
761 specific operation. A KNX IoT device, therefore, SHALL support the following command to allow a
762 MaC, or any other client with appropriate rights, to reset the KNX IoT device to its default configuration
763 state.

764 Table 11 defines MANDATORY ("M") resources of a KNX IoT device and the corresponding resource
765 path names that SHALL be used.

766 **Table 11 – Device Reset Command**

Resource path	rt & Data Types	Format	Method	Support	Request/Response	Notes
/.well-known/knx	NA	cbor	POST	**M**	Req: Content-Format: application/cbor Payload: { 2: "reset", 1: <erase-code> } Res: Content-Format: application/cbor Payload: { "code": <error-code>, "time": <process-time> }	The CoAP reset request SHALL be confirmed by the device before the CoAP connection breaks down.

767

768 To perform a Master Reset, the KNX IoT device SHALL reset its configuration data according to the
769 following:

770 • The Function Point Tables SHALL be cleared. Hence the Master Reset stops runtime
771 interworking (group communication).

772 • Depending on the erase code, any network and security parameters SHALL be reset to their ex-
773 factory default state (applicable for erase codes 0x2 and 0x3, but explicitly not for erase code
774 0x7); with erase-code 0x7, all security parameters (auth/at, auth/crts) SHALL be reset, except the
775 parameters under access scope "if.sec".

776 • A (Basic) Restart is executed.

For the definition of erase codes, see [09] Table 4. From the list of the erase codes defined there, a KNX IoT device SHALL support erase codes 0x2 and 0x7. It MAY support further erase codes, e.g., 0x3, for explicitly resetting the network and security parameters.

Before executing the reset function, the KNX IoT device SHALL return a response with CoAP response code "2.04 Changed" and with a payload containing Error Code and Process Time in seconds as defined for the response to a Master Reset Request for KNX Classic devices, see [09]. The following example shows how a MaC sets a KNX IoT device to its default configuration state.

(Master) Reset to the default state.
REQ: POST coap://{ipv6-unicast}/.well-known/knx (Content-Format: application/cbor (60)) **Payload:** { 2: "reset", 1: <erase-code> } **RES:** 2.04 CHANGED (Content-Format: application/cbor (60)) **Payload:** { "code": <error-code>, "time": <process-time> }

2.5.5.3.3 (Basic) Restart Command

A KNX IoT device SHALL support the following command to allow a MaC, or any other client with appropriate rights, to restart a KNX IoT device.

Table 12 defines MANDATORY ("M") resources of a KNX IoT device and the corresponding resource path names that SHALL be used.

Table 12 – Device Restart command

Resource path	rt & Data Types	Method	Support	Request/Response	Notes
/.well-known/knx	NA	POST	**M**	Req: Content-Format: application/cbor Payload: { 2: "restart" }	The device SHALL NOT confirm the CoAP restart request, and the restart may result in a breakdown of the CoAP connection.

To perform a (Basic) Restart, the KNX IoT device SHALL

- switch off Programming Mode
- terminate the validity of the responder key of an open PASE session (see 3.6.3)
- apply changed configuration parameters at the latest 30 s after completing the restart.

It is explicitly not required but optional to restart the network stack or the overall hardware/firmware of the device. Consequently, a restart may result in a breakdown of the CoAP connection.

2.5.5.4 Device Configuration Fingerprint (.well-known/knx/f)

A KNX IoT device SHALL provide a resource to read a fingerprint of the device configuration settings (e.g., a checksum). The device configuration fingerprint SHALL include parameters that can be changed of the following resources:

- /{base-path}/fp/g
- /{base-path}/fp/r
- /{base-path}/fp/p
- /{base-path}/p

A MaC MAY uses the device configuration fingerprint to decide wheater a differential download is possible or not. The calculation of the device configuration fingerprint is manufacturer specific but SHALL reflect distinct device configuration settings. The KNX IoT device SHALL reply with the configuration fingerprint only in "loaded" state.

Suppose the KNX IoT device cannot immediately respond since it is busy (e.g., status ≠ loaded); the device SHALL return response code "5.03 Service Unavailable" but uses the Max-Age option to indicate the number of seconds after which to retry.

Table 13 defines MANDATORY ("M") resources of a KNX IoT device and the corresponding resource path names that SHALL be used.

Table 13 – Device Configuration Fingerprint resource

Resource path	rt & Data Types	Method	Support	Request/Response	Notes
/.well-known/knx/f	NA	GET	**M**	Req: Content-Format: application/cbor Payload: { 1: <value> }	Device-specific fingerprint to identify changes in the resources "fp/g", "fp/r", "fp/p", and "p".

2.5.5.5 Device Individualization Resource (.well-known/knx/ia)

Each KNX IoT device, i.e., a Router or an end device, SHALL have a unique *KNX Individual Address* in a network. The *KNX Individual Address* is a 2-octet value that consists of an 8-bit Subnetwork Address (see "dev/sna") and an 8-bit Device Address (see "dev/da").

KNX Individual Address	
Octet 0	Octet 1
Subnetwork Address	Device Address

Example: Subnetwork Address: 0, Device Address: 1 = 0x0001

The individualization process is derived from the A_IndividualAddress_Write-service defined for KNX Classic devices (see [11]). However, KNX IoT splits the functionality into a *Programming Mode* discovery (see clauses 2.6.1.2.3 and 2.6.1.3.6.3) and a device address configuration step.

The "/.well-known/knx/ia" resource SHALL be used by the MaC to modify the *KNX Individual Address* and the installation ID on a KNX IoT device (see clause 2.4.2).

The device individualization request SHALL contain the new *KNX Individual Address* and installation ID. Table 14 defines the MANDATORY ("M") resource object members and CBOR keys. If a member is marked as OPTIONAL ("O"), a client can omit the member in the request. However, the server SHALL support the member handling.

832 **Table 14 – Device Individualization Resource Object**

JSON Key	CBOR Key	CBOR Type	Support	Description
"ia"	12	unsigned	**M**	New KNX Individual Address device configuration (2 Byte).
"fid"	25	unsigned	O	New KNX Fabric ID device for unicast address configuration.
"iid"	26	unsigned	**M**	New KNX Installation ID for device multicast configuration.

833

834 Supposed the KNX IoT device has changed its local *KNX Individual Address* (see "dev/sna", "dev/da",
835 "dev/iid", and "dev/fid" in clause 2.5.6), then the KNX IoT device SHALL return a response with CoAP
836 response code "2.04 Changed". After responding to the client, the KNX IoT device SHALL change the
837 Ipv6 address if the KNX Fabric ID has changed (see ULA clause 2.6.2).

838 Table 15 defines MANDATORY ("M") resources of a KNX IoT device and the corresponding resource
839 path names that SHALL be used.

840 **Table 15 – Device Individualization Resource**

Resource path	rt & Data Types	Method	Support	Request/Response	Notes
/.well-known/knx/ia	NA	POST	**M**	Req: Content-Format: application/cbor Payload: { 12: <*KNX Individual Address*>, 26: <iid> }	Device individualization for configuring the device *KNX Individual Address* and installation ID.

841

842 ### 2.5.6 Device Object Resource (dev)

843 The device object contains general device-specific settings and configurations. The MaC configures some
844 Properties (see clause 2.4.3, "Device Configuration Procedure"), and other Properties are configured in
845 the factory (e.g., *KNX Serial Number*).

846 Table 16 specifies the MANDATORY ("M") or OPTIONAL ("O") device resources (dev) and the
847 corresponding resource path names that SHALL be used.

848 **Table 16 – Device resources and paths**

Resource path	rt & Data Types	Format	Method	Support	Request/Response	Notes
dev	:fb.0	link-format	GET	**M**	Content-Format: application/link-format Payload: </dev/sn>;rt=":dpa.0.11";ct=60, </dev/fwv>;ct=60, </dev/hwt>;ct=60, </dev/model>;rt=":dpa.0.15";ct=60, </dev/sna>;rt=":dpa.0.57" ;ct=60, </dev/da>;rt=":dpa.0.58>";ct=60, </dev/hname>;ct=60, </dev/ipv6>;ct=60, </dev/fid>;ct=60, </dev/iid>;ct=60, </dev/mport>;ct=60, </dev/pm>;rt=":dpa.0.54";ct=60	Device *Functional Block*.
dev/sn	:dpa:0.11 :dpt.serNum	cbor (json)	GET	**M**	Content-Format: application/cbor Payload: { 1: "001caffe1234" }	Device *KNX Serial Number.*
dev/mid	:dpa:0.12 :dpt.value2Ucount	cbor (json)	GET	**M**	Content-Format: application/cbor Payload: { 1: 0x00fa }	Manufacturer Identifier
dev/hwv	:dpt.version	cbor (json)	GET	O	Content-Format: application/cbor Payload: { 1: [1,2,3] }	Hardware version.
dev/fwv	:dpa.0.25 :dpt.version	cbor (json)	GET	**M**	Content-Format: application/cbor Payload: { 1: [1,2,3] }	Firmware version.
dev/hwt	:dpt.varString8859_1	cbor (json)	GET	**M**	Content-Format: application/cbor Payload: {1: "0123AB"}	The hardware type is a manufacture-specific ID for a device type (MaC uses this ID for compatibility checks). Maximum length: 6 octets.
dev/model	:dpa.0.15 :dpt.utf8	cbor (json)	GET	**M**	Content-Format: application/cbor Payload: { 1: "QAA987" }	Device Model Maximum length: 10 octets.

Resource path	rt & Data Types	Format	Method	Support	Request/Response	Notes
dev/sna	:dpa.0.57 :dpt.value1Ucount	cbor (json)	GET	**M**	Content-Format: application/cbor Payload: { 1: 1 }	KNX Subnet Address. Default: 255 (see [10])
dev/da	:dpa.0.58 :dpt.value1Ucount	cbor (json)	GET	**M**	Content-Format: application/cbor Payload: { 1: 2 }	KNX Device Address. Default: 255 (see [10])
dev/hname	:dpt.varString8859_1	cbor (json)	GET PUT	**M**	Content-Format: application/cbor Payload: {1: "knx-001caffe1234.local" }	Device hostname for DNS resolution (minimum 60 and maximum 253 characters according to [RFC1035] clause 2.3.4).
dev/ipv6	:dpt.ipv6	cbor (json)	GET (PUT)	**M**	Content-Format: application/cbor Payload: { 1: h'20010db8000300006cd98ad28e881f68'} or Content-Format: application/cbor Payload: [{1: h'20010db8000300006cd98ad28e881f68'}, {1: h'fde58dba82e10001000000fffe000401'}]	Array of device ipv6 addresses. The resource interface SHALL be "if=if.b" (see clause 2.5.3.8).
dev/fid	:dpt.value8Ucount	cbor (json)	GET (PUT)	**M**	Content-Format: application/cbor Payload: { 1: 0x11222233a3 }	40-bit KNX Fabric ID (see ULA in clause 2.6.2). Maximum: 0x00ff'ffff'ffff
dev/iid	:dpt.value8Ucount	cbor (json)	GET (PUT)	**M**	Content-Format: application/cbor Payload: { 1: 0x00000033a3 }	40-bit KNX Installation ID (see ULA in clause 2.6.5). Maximum: 0x00ff'ffff'ffff
dev/port	:dpt.value2Ucount	cbor (json)	GET (PUT)	**M**	Content-Format: application/cbor Payload: { 1: 5683 }	Port number used for unicast comm. (see clause 2.6.3.1).

Resource path	rt & Data Types	Format	Method	Support	Request/Response	Notes
dev/mport	:dpt.value2Ucount	cbor (json)	GET	**M**	Content-Format: application/cbor Payload: { 1: 5683 }	Port number used for multicast discovery (see clause 2.6.4.1.)
dev/pm	:dpa.0.54 :dpt.binaryValue	cbor (json)	GET PUT	**M**	Content-Format: application/cbor Payload: { 1: false }	Programming Mode enable/disable.

2.5.7 Function Point Table (fp)

2.5.7.1 Introduction and Overview

This clause describes the configuration of group events on a KNX IoT device. A MaC can choose between several message exchange configurations. The definition of when to apply which message exchange pattern is out-of-scope of this document. A KNX IoT device SHALL support at least the following message exchange configurations:

- Multicast message exchange
- Direct unicast message exchange

A KNX IoT device MAY support, and the KNX IoT Router SHALL support, the following message exchange configuration:

- Endpoint subscription message exchange (see [12] "KNX IoT Router")

In Figure 8, the MaC configures a multicast message exchange, for example, between one or many sensors (e.g., light switches) and one or many actuators (e.g., light actuators). In addition, the actuator status is sent to an HMI (e.g., room unit) for visualization. For multicast group events, the MaC SHALL configure the following Function Point Tables:

- GA and Points in the Group Object Table on all devices.
- Multicast send address in the Function Point Recipient Table on devices with "if.o" interfaces.
- Multicast listening address in the Function Point Publisher Table on devices with "if.i" interfaces.

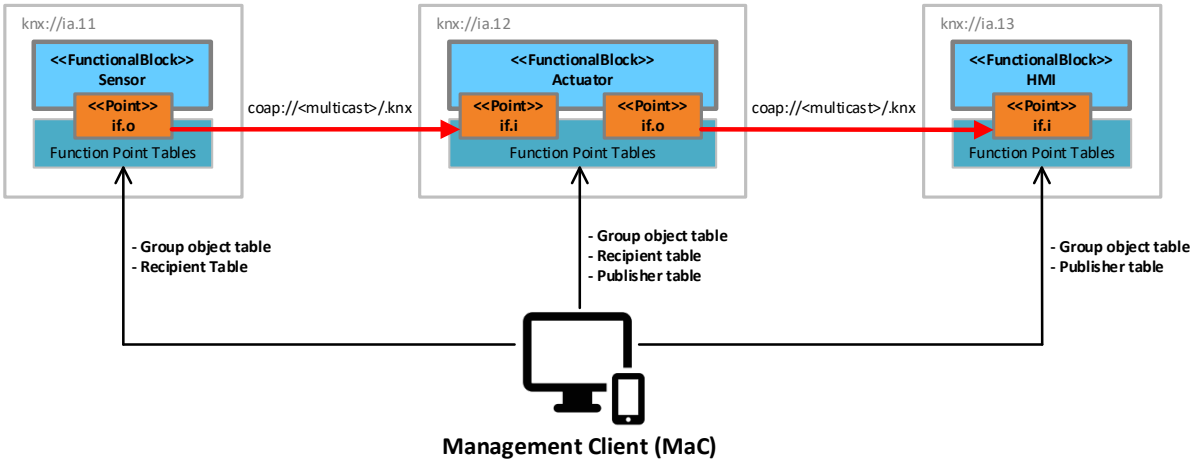


Figure 8 – Multicast Message Exchange

In the next example, the MaC configures direct unicast message exchange in networks where multicast reachability between devices is impossible or for applications that need reliable (confirmed) communication between devices. Figure 9 depicts a unicast example with a controller (e.g., HVAC controller) which gets sensor updates (e.g., room temperature sensor) and commands an actuator accordingly (e.g., radiator valve). The MaC SHALL configure for unicast group events the following Function Point Tables:

- GA and Points in the Group Object Table on all devices.
- Unicast destination address in the Function Point Recipient Table on devices with "if.o" interfaces (*KNX Individual Address* 21 and 22 are sending messages to *KNX Individual Address* 22 and 23).
- No Function Point Publisher Table configuration on all devices.

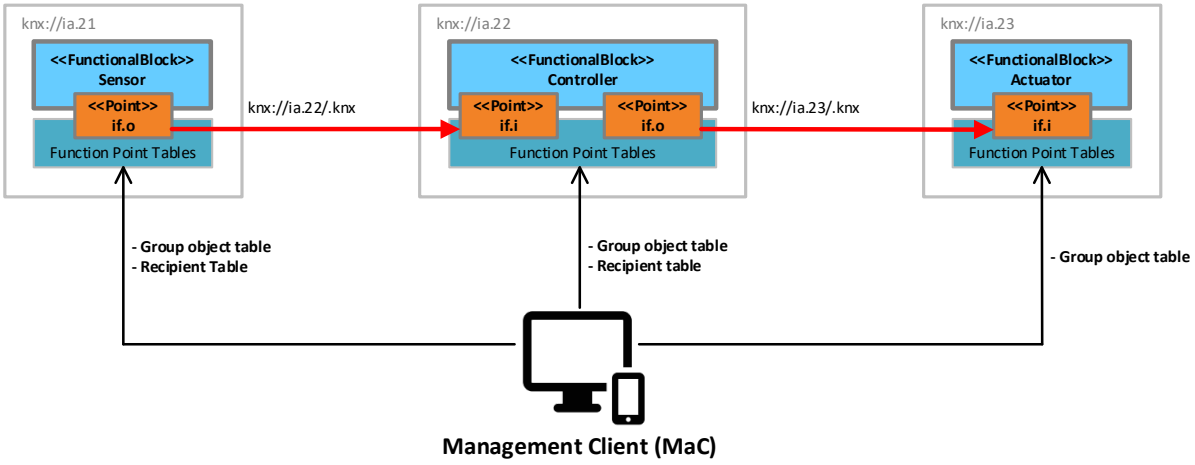


Figure 9 – Direct Unicast Message Exchange

2.5.7.2 Device Load State Machine State

If a CoAP request method (GET, PUT, or POST) to Group Object Table, Function Point Recipient Table, or Function Point Publisher Table items or collections is possible depends on the current Device Load State Machine state (see clause 2.5.8). CoAP GET SHALL be possible in all Device Load State Machine states. However, in states loaded, unloaded, unloading, and loadcompleting (state ≠ loading), the KNX IoT device SHALL return response code "4.05 Method Not Allowed" to PUT, POST, and DELETE requests.

2.5.7.3 Group Object Table Resource (fp/g)

2.5.7.3.1 Definition

A KNX IoT device SHALL implement a Group Object Table. The Group Object Table SHALL be writable; it is configured by the MaC. The Group Object Table contains configuration information for each *Group Object* used by the device.

A *Group Object* is represented by a manufacturer-specific resource path ("href": "{point-path}"). A *Group Object* has a set of mandatory and optional metadata members (see clause 2.5.11.3, "Metadata Query Parameter "m" and clause 2.5.11.4, "Metadata Resource Object"). A Metadata member is accessible as a resource behind the {point-path}. A MaC SHALL use metadata members such as CoV, heartbeat, min/max limitations, etc., to configure *Group Object* communication behavior.

If a function has extended features with additional configuration parameters, then the MaC MAY write values to other manufacturer-specific resource paths ({point-path}). This document does not specify how to implement manufacturer-specific Parameters and Diagnostic Properties.

Every *Group Object* SHALL implement a ("dpt") metadata for retrieving Datapoint Types and, if possible, semantic Datapoint Annotations ("rt").

KNX Datapoint Types are identified by a prefix (urn:knx:dpt) and a Datapoint Type separated by a dot as specified in [03] "KNX Information Model". For example, a DPT_BinaryValue is encoded as "urn:knx:dpt.binaryValue". Datapoint Types with the same main number have the same data type (e.g., integer or boolean), format (e.g., B1 or B2), and encoding. A different subnumber indicates a different dimension (different range and/or different unit). KNX Datapoint Types are defined in KNX System specifications, chapter 3/7/2, and are used accordingly in KNX IoT. The Datapoint Type SHALL be defined for every *Group Object* and can be retrieved by reading {point-path}?m=dpt.

All *Group Objects* SHOULD have a semantic Datapoint Annotation, for example, for filtering discovery requests. Datapoint annotations have the following structure: urn:knx:dpa.{functionalblock-id>.<property-id>. Datapoint Annotations are inherited from the existing *Functional Block* and Property specification defined in the KNX Application Descriptions (KNX standard Volume 7). A Datapoint Annotation does not imply that a specific *Functional Block* functionality is implemented. Every *Group Object*, or rather the assigned {point-path}, SHOULD have a Datapoint Annotation unless the appropriate *Functional Block* and Property has not yet been defined in the KNX Application Descriptions.

A *Group Object* contains a list of *Group Addresses* ("ga"). However, only one sending *Group Address* can be used for a *Group Object*. Hence, if more than one *Group Address* is configured, then the first list item SHALL be used as sending *Group Address*. In addition, if multiple *Group Objects* are linked to the same Point ("href"), then the *Group Object* with the lowest "id" SHALL contain the sending *Group Address*.The "href" member SHALL reference a {point-path}.

Table 17 specifies the MANDATORY ("M") or OPTIONAL ("O") Group Object Table resources and the corresponding resource path names that SHALL be used.

Table 17 – Group Object Table resources and paths

Resource path	rt & Data Types	Format	Method	Support	Request/Response	Notes
fp/g	NA	cbor	POST	M	Req: Content-Format: application/cbor Payload: (see example below)	Write Group Object Table. Single Elements are identified by their "id". Existing elements are replaced, and not existing elements are added. An empty element (only "id") in the payload will cause the deletion of a corresponding existing element.
fp/g	NA	cbor	GET	**M**	Resp: Content-Format: application/link-format Payload: </fp/g/1>;ct=60	Group object item linked list.
fp/g/{group-object-id}	NA	cbor	GET DELETE	**M**	Req: Content-Format: application/cbor Payload: (see example below)	Read/delete single list item. The group-object-id corresponds to the resource object "id" value as a number in ASCII format.

926 2.5.7.3.2 Function Point Group Object Resource Object

927 Table 18 defines the MANDATORY ("M") Function Point Group Object resource object CBOR keys for
928 configuring and reading Group Object Table items. MANDATORY ("M") attributes SHALL be present
929 in the message.

930 **Table 18 – Function Point Group Object resource object CBOR keys**

JSON Key	CBOR Key	CBOR Type	Support	Description
"id"	0	unsigned	**M**	The ID of the Group Object Table Entry. It has to be unique in the table (range 0…65 535). Note: There is no sorting requirement for IDs.
"ga"	7	unsigned array	**M**	A List of all connected *Group Addresses*. The Group Object Table SHALL support >= 20 array elements.
"cflag"	8	unsigned	**M**	Group Object Configuration Flags (see 2.5.7.3.3).
"href"	11	text string	**M**	Reference to the Point, which represents the data of this Group Object.

931

932 In the following example, the MaC configures two Group Object Table items on a KNX IoT device. The
933 "cflag" for the first item is configured as "read", "write", "transmit", and "update" (0b11011000). The
934 "cflag" for the second item is configured as "transmit" (0b01000000).

935

Write Group Object Table items.

REQ:
POST coap://{ipv6-unicast}/fp/g
(Content-Format: application/cbor (60)), OSCORE(code(POST), kid(<osc-id>)))
Payload:

```
[
  {
    0: 1,
    11: "/p/ldsb1/soo",
    7: [2305, 2401],
    8: 216                    //0b11011000
  },
  {
    0: 2,
    11: "/p/ldsb1/rsc",
    7: [2306],
    8: 64                     //0b01000000
  }
]
```

936 #### 2.5.7.3.3 Group Object Configuration Flags Resource Object "cflag"

937 A KNX IoT device SHALL support for each *Group Object* the non-volatile configuration flags "read",
938 "init", "write", "transmit", and "update", describing the communication behavior of the corresponding
939 *Group Object*. Like in KNX Classic, the configuration flags are configured by a MaC and are written to
940 the device as part of the Group Object Table configuration, see 2.5.7.3 "Group Object Table".

941 The meaning and action of the configuration flags "read", "init", "write", "transmit", and "update" are
942 explained in Table 19.

943 **Table 19 – Group Object configuration flags**

Flag	Bit	Description
Reserved	0, 1	Default = false
Communication	2	Not used in KNX IoT since the flag is just a logical disjunction (OR) of the other configuration flags (default = false).
Read	3	false = *Group Object* value cannot be read. true = With a read command to a *Group Object*, the device sends this *Group Object's* value.
Write	4	false = *Group Object* value cannot be written. true = The device receives and overwrites its *Group Object* value (default for inputs).
Init (Value read on initialization)	5	false = Disable read after initialization. true = *Group Object* read request after initialization (e.g., power up). The read-Flag on the producer side is expected to be true to get a response.
Transmission	6	false = *Group Object* value is not transmitted. true = For CoV, event, or heartbeat, the device sends its *Group Object* value (default for outputs).
Update	7	false = *Group Object* value is not updated. true = The *Group Object* value is updated with a group value response message if flag w=true.

944 #### 2.5.7.3.4 Group Object Table Modifications

945 A MaC SHALL create or update *Group Object* items by sending a CoAP POST request to /{base-
946 path}/fp/g, which contains the KNX IoT device *Group Object* configuration (see clause 2.6.7, Creating,
947 Updating, and Deleting Resources).

948 A new item SHALL overwrite the entire item (incl. "ga", "cflag", etc.) in the case an item with the same
949 "id" already exists on the device.

950 However, the MaC SHALL NOT configure multiple *Group Object* items with the same "href" Point
951 reference but different "cflags" settings, as shown in the figure below, since this functionality is not
952 supported in KNX Classic.

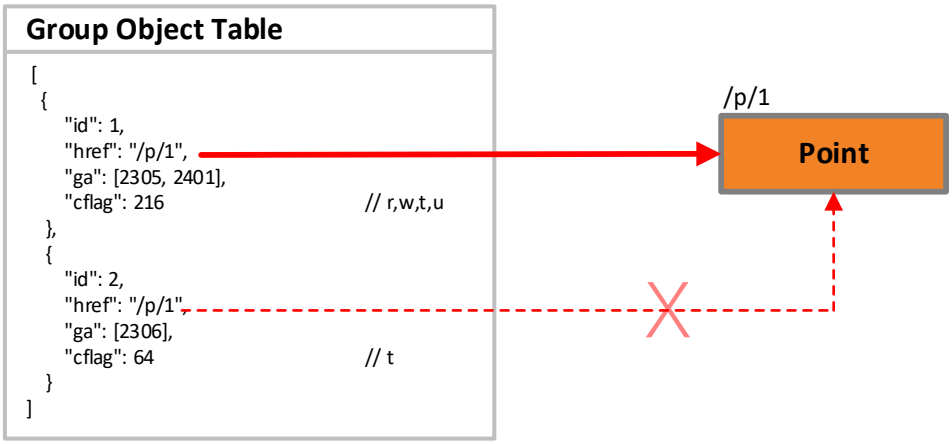


Figure 10 – Group Object Table Misconfiguration

2.5.7.4 Function Point Recipient Table Resource (fp/r)

2.5.7.4.1 Definition

The Function Point Recipient Table contains the destination address of outgoing group events (see clause 2.3.4.2, "Function Point Tables"). This table SHALL be writable and is configured by the MaC. The Function Point Recipient Table SHALL support the following endpoint configurations:

- IPv6 unicast destination address of a KNX IoT Recipient device that provides a CoAP endpoint.
- or a *KNX Individual Address* or FQDN that can be resolved to an IPv6 unicast address of a KNX IoT Recipient device.
- or IPv6 multicast addresses to send a message to multiple KNX IoT Recipients.

Table 20 specifies the MANDATORY ("M") resources and the corresponding resource path names that SHALL be used for the Function Point Recipient Table.

Table 20 – Function Point Recipient Table resource path and methods

Resource path	rt & Data Types	Format	Method	Support	Request/Response	Notes
fp/r	NA	cbor	POST	**M**	Req: Content-Format: application/cbor Payload: (see example below)	Write Function Point Recipient Table. Single elements are identified by their "id". Existing elements are replaced, and not existing elements are added. An empty element (only "id") in the payload will cause the deletion of a corresponding existing element.
fp/r	NA	cbor	GET	**M**	Res: Content-Format: application/link-format Payload: </fp/r/1>;ct=60	Recipient item linked list

Resource path	rt & Data Types	Format	Method	Support	Request/Response	Notes
fp/r/{recipient-id}	NA	cbor	GET DELETE	**M**	Req: Content-Format: application/cbor Payload: (see example below)	Read/delete single list item. The recipient-id corresponds to the resource object "id" value as a number in ASCII format.

967

968 2.5.7.4.2 Function Point Recipient Resource Object

969 A Function Point Recipient Resource Object represents either a direct unicast or a multicast message
970 exchange which defines an IPv6 endpoint configuration. The IPv6 address and the path are used as the
971 destination endpoint address in the CoAP message.

972 A KNX IoT device MAY publish messages to topics. The topic name structure in the superordinate
973 system is often given, such as "kitchen/sensors/". The destination address (URL including path) in the
974 Function Point Recipient Table SHALL be freely configurable. The MaC SHALL configure the topic
975 name in the Recipient Table (see examples in clause 2.3 "System Design") only in cases where the
976 endpoint path deviates from the standard KNX path (.knx).

977 The MaC can change some basic message transmission parameters with "non" and "mt" since some
978 applications MAY need adjusted settings. It is RECOMMENDED that KNX IoT devices use CoAP
979 default settings as defined in [RFC7252] clause 4.8. Manufacturers MAY support additional
980 configuration parameters conveyed as proprietary metadata parameters (see clause 2.5.11.3, "Metadata
981 Query Parameter "m"). However, this is out-of-scope of this document.

982 Table 21 defines the Function Point Recipient resource object CBOR keys for configuring and reading
983 Function Point Recipient Table items. MANDATORY ("M") attributes SHALL be present in the
984 message, CONDITIONAL ("C") attributes SHALL be present in some special cases and OPTIONAL
985 ("O") MAY are present in the message, but the attributes SHALL be supported on the device.

986 **Table 21 - Function Point Recipient resource object CBOR keys**

JSON Key	CBOR Key	CBOR Type	Support	Description
"id"	0	unsigned	**M**	The ID of the Function Point Recipient Table entry. It has to be unique in the table.
"ga"	7	unsigned array	**M**	A list of *Group Addresses* which to be sent to the configured Recipient. The Function Point Recipient Table SHALL support >= 20 array elements. A device SHALL accept empty “ga” arrays for the Function Point Recipient Table.
"url"	10	text string	O[a]	The URL identifies the Recipient endpoint. The CoAP URI schema SHALL be supported.

JSON Key	CBOR Key	CBOR Type	Support	Description
"ia"	12	unsigned	O[a]	KNX Individual Address of the Recipient.
"grpid"	13	unsigned	C[a]	Group ID for the multicast group identifier (see clause 2.6.5) for sending multicast messages to the network. The "grpid" SHALL be used for the multicast group identifier.
"at"	14	text string	C[a]	Access token id. Reference to the security credentials for message encryption.
"fid"	25	unsigned	C[a]	KNX Fabric ID of the Recipient. Not used in case Recipient has the same fabric ID (see clause 2.6.2 "Device IP Address").
"iid"	26	unsigned	C[a]	KNX Installation ID of the Recipient. Not used in case Recipient is within the same installation (2.6.5 Multicast Group IP Addresses)
"non"	"non"	bool	O	true = Non-confirmable request. If "non" is omitted, the device SHALL assume a confirmed request (false) as default for unicast messages. Multicast messages SHALL be Non-confirmable (true).
"mt"	"mt"	unsigned	O	The number of maximum retransmission for confirmable and non-confirmable requests. If "mt" is omitted, the device SHALL assume no repetition (0) for non-confirmable requests and a maximum of four (4) retransmissions for confirmable requests. However, the device SHALL send not more than five (5) retransmissions.
[a] The Recipient can be identified either by "ia", "grpid", or by "url".				

987

2.5.7.4.3 Publish Group Events

Table 22 describes the basic procedure for sending group event messages to Recipients and how the Function Point Tables are used.

Table 22 – Publish Group Event Messages

Step	Procedure	Description
1	Check Point value.	- Check if a Point value has changed AND time since the last update is > min repetition time - OR time since last update > heartbeat (see clause 2.5.11.2)
2	Check group object configuration flags.	- KNX IoT device checks the group object configuration flags (see clause 2.5.7.3.3) if the corresponding Point value needs to be published.
3	Check Recipient Table items.	- The KNX IoT device searches Group Object Table items and Functional Point Recipient Table items with the same *Group Address* ("ga") configuration.
4	Send group message to Recipient.	- If the *Group Address* ("ga") configuration matches with a Functional Point Recipient Table item, then the KNX IoT device sends a group message to the network in the case: o Functional Point Recipient Table item is configured with a valid multicast address ("grpid" or "url"). o Functional Point Recipient Table item is configured with a valid unicast address ("ia" or "url").
5	Send group message to Subscriptions.	- If the Group Object Table contains a *Group Address* ("ga") configuration and a CoAP client has subscribed to ".knx" (CoAP observe), then the device SHALL send a notification to subscribed CoAP clients.
6	Update the last sent timestamp.	- If the message was sent to the network, the device SHALL update the last sent timestamp. This step is only applicable if heartbeat or minimum repetition timeout is configured.

Figure 11 shows a basic multicast Recipient configuration. Multicast group event notifications do not need additional IPv6 address resolving steps or acknowledgments; potential repetitions are also unnecessary. Therefore, the sending procedure of a multicast Recipient configuration is straightforward and follows the steps in Table 22.

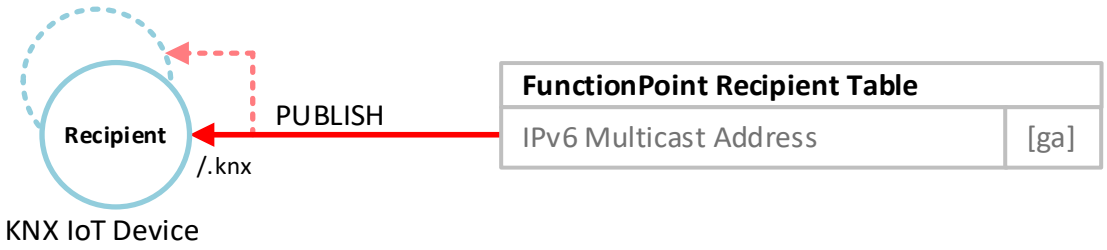


Figure 11 – Multicast Function Point Recipient Table Item

The following example configures the multicast Recipient item from Figure 11 with the default path ".knx".

The "grpid" attribute SHALL be used for multicast Recipient configuration, and the "ia" SHALL be used for unicast Publisher configuration. The installation ID is needed for the Recipient IPv6 multicast address since the *Group Address* is unique only within an installation but can be omitted if the Recipient belongs to the same installation.

The "url" attribute can be used for unicast as well as for multicast Publisher but SHALL NOT be used in combination with "ia" or "grpid".

Function Point Recipient Table Configuration Example.

```
REQ:
POST coap://{ipv6-unicast}/fp/r
(Content-Format: application/cbor (60), OSCORE(code(POST), kid(<osc-id>)))
Payload:
[
   {
      0: 2,
      26: <recipient's installation ID>,
      13: <multicastGroupId>,
      14: <access token ID>,
      7: [2204, 2205, 2206]
   },
   {
      0: 3,
      10: "coap://<IP multicast>:<port>/<path>",
      14: <access token ID>,
      7: [2305, 2306, 2307, 2308]
   },
   {
      0: 4,
      13: <multicastGroupId>,
      14: <access token ID>,
      7: [2404, 2405, 2406]
   }
]
```

A MaC MAY configure a unicast communication between KNX IoT devices. The MaC can configure unicast communication with a fixed IPv6 address, a *KNX Individual Address,* or an FQDN. In the case of a *KNX Individual Address* or an FQDN configuration, the KNX IoT device has to resolve the *KNX Individual Address* or FQDN to an Ipv6 unicast address before the device can publish events.

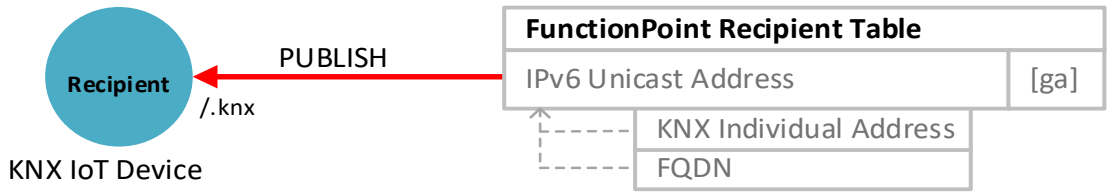


Figure 12 – Unicast Function Point Recipient Table Item

The following example shows a Function Point Recipient Table configuration that uses an FQDN and a *KNX Individual Address* as destination endpoint configuration. The KNX Fabric ID "fid" SHALL be omitted if the Recipient belongs to the same project.

Function Point Recipient Table Configuration Example.

REQ:

```
POST coap://{ipv6-unicast}/fp/r
(Content-Format: application/cbor (60), OSCORE(code(POST), kid(<osc-id>)))
```

Payload:

```
[
  {
    0: 5,
    25: <recipient's fabric ID>,
    12: <recipient's IA>,
    14: <access token ID>,
    7: [2505, 2506, 2507, 2508]
  },
  {
    0: 6,
    10: "coap://<unicast address or fqdn>:<port>/<path>",
    14: <access token ID>,
    7: [2605, 2606, 2607, 2608]
  },
  {
    0: 7,
    12: <recipient's IA>,
    14: <access token ID>,
    7: [2705, 2706, 2707, 2708]
  }
]
```

2.5.7.5 Function Point Publisher Table Resource (fp/p)

2.5.7.5.1 Definition

The Function Point Publisher Table contains the source address of incoming group events (see clause 2.3.4.2, "Function Point Tables"). This table SHALL be writable and is configured by the MaC. The Function Point Publisher Table supports the following endpoint configurations:

- IPv6 unicast subscription address of a KNX IoT Publisher device.
- or IPv6 multicast addresses to receive messages from multiple KNX IoT Publishers.

Table 23 specifies the MANDATORY ("M") resources and the corresponding resource path names that SHALL be used for the Function Point Publisher Table.

1027 **Table 23 – Function Point Publisher table**

Resource path	rt & Data Types	Format	Method	Support	Request/Response	Notes
fp/p	NA	cbor	POST	**M**	Req: Content-Format: application/cbor Payload: (see example below)	Writes the Function Point Publisher Table. Single elements are identified by their "id". Existing elements are replaced, and not existing elements are added. An empty element (only "id") in the payload will cause the deletion of a corresponding existing element.
fp/p	NA	cbor	GET	**M**	Res: Content-Format: application/link-format Payload: </fp/p/1>;ct=60	Publisher item linked list
fp/r/{publisher-id}	NA	cbor	GET DELETE	**M**	Req: Content-Format: application/cbor Payload: (see example below)	Read/delete single list item. The publisher-id corresponds to the resource object "id" value as a number in ASCII format.

1028

1029 #### 2.5.7.5.2 Function Point Publisher Resource Object

1030 A Function Point Publisher Resource Object represents a multicast or a subscription message exchange.
1031 The IPv6 address is used as a unicast IPv6 subscription endpoint or a receiving multicast IPv6 address
1032 configuration.

1033 A KNX IoT device SHALL support multicast message exchange configurations and MAY support
1034 subscription (unicast address) configurations in the Function Point Publisher Table. However, a KNX IoT
1035 device SHALL support subscriptions on the resource ".knx" as defined in clauses 2.5.9.3 and 2.5.9.4. This
1036 allows clients, such as a MaC or Message Broker, to subscribe for event notifications on all KNX IoT
1037 devices.

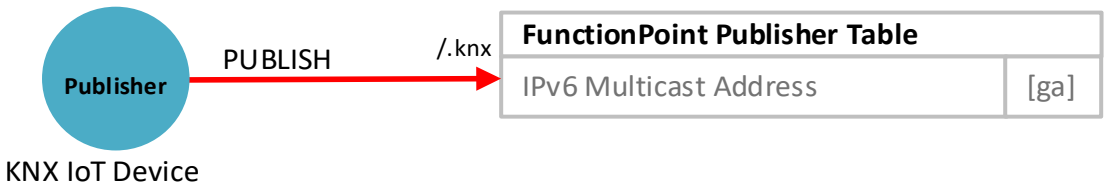
1038 Table 24 defines the Function Point Publisher resource object CBOR keys for configuring and reading
1039 Function Point Publisher Table items. MANDATORY ("M") attributes SHALL be present in the
1040 message. CONDITIONAL ("C") attributes SHALL be present in some special cases and OPTIONAL
1041 ("O") maybe present in the message, but the attributes SHALL be supported on the device.

1042 **Table 24 – Function Point Publisher resource object**

JSON Key	CBOR Key	CBOR Type	Support	Description
"id"	0	unsigned	**M**	The ID of the Function Point Publisher Table entry. It SHALL be unique in the table.
"ga"	7	unsigned array	**M**	A list of *Group Addresses* that SHALL be received from the configured Publisher. The Function Point Publisher Table SHALL support >= 20 array elements. A device SHALL accept empty "ga" arrays for the Function Point Publisher Table.
"url"	10	text string	C[a]	The URL identifies the Publishers endpoint for unicast communication. The CoAP URI schema SHALL be supported.
"ia"	12	unsigned	C[a]	The KNX Individual Address of the Publisher for unicast communication (see 2.5.5.5).
"grpid"	13	unsigned	C[a]	Group ID for the multicast group identifier (see clause 2.6.5). Used for multicast listening on the network.
"at"	14	text string	C[a]	Access token id. Reference to the security credentials for unicast subscription encryption.
"fid"	25	unsigned	C[a]	KNX Fabric ID of the Publisher. Not used in case the Publisher has the same KNX Fabric ID (see also clause 2.6.2 "Device IP Address").
"iid"	26	unsigned	C[a]	KNX Installation ID of the Publisher. Not used in case the Publisher is within the same installation (2.6.5 Multicast Group IP Addresses)

[a] The Publisher can be identified either by "ia", "grpid", or "url".

1043
1044 Figure 13 shows a KNX IoT device listening on an IPv6 multicast address. The Publisher sends a
1045 multicast event notification to the network. Conversely, a KNX IoT device receives the event
1046 notifications and updates from a configured *Group Address* or, rather, on the associated Point in the
1047 Group Object Table.



1048
1049 **Figure 13 – Multicast Function Point Publisher Table Item**

The MaC MAY configure on a KNX IoT device a subscription endpoint (Publisher) with a project-specific resource endpoint (path). The resource path on the Publisher in a superordinate system is often given, for example, "kitchen/sensors/". Therefore, the KNX IoT device SHALL support project-specific subscription endpoints in the Function Point Publisher Table.

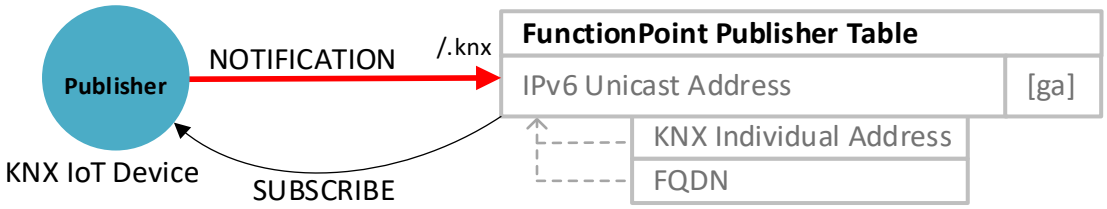


Figure 14 – Unicast Function Point Publisher Table Item

A KNX IoT device MAY have to subscribe (see [12] "KNX IoT Router") or send group events to a device that belongs to another KNX IoT installation. The KNX Fabric ID "fid" is needed for resolving the Publisher Ipv6 unicast address since the *KNX Individual Address* is unique only within an installation (see clause 2.6.1.2, "Device Discovery with DNS-SD"). The KNX Fabric ID SHALL be omitted if the Publisher belongs to the same fabric.

The "ia" attribute SHALL be used for unicast Publisher configuration and the "grpid" for multicast Publisher configuration. The installation ID is needed for the Publisher Ipv6 multicast address since the *Group Address* is unique only within an installation.

The "url" attribute can be used for unicast as well as for multicast Publisher but SHALL NOT be used in combination with "ia" or "grpid".

Function Point Publisher Table Configuration Example.

REQ:
POST coap://{ipv6-unicast}/fp/p
(Content-Format: application/cbor (60), OSCORE(code(POST), kid(<osc-id>)))
Payload:

```
[
  {
    0: 7,
    25: <publisher fabric ID>,
    12: <publisher's IA>,
    14: <access token ID>,
    7: [2305, 2306, 2307, 2308]
  },
  {
    0: 8,
    10: "coap://<IP multicast, unicast address or fqdn>:<port>/<path>",
    7: [2309, 2310, 2311, 2312]
  },
  {
    0: 9,
    26: <publisher installation ID>,
    13: <multicastGroupId>,
    7: [2320, 2321, 2322, 2323]
  }
]
```

2.5.8 Application Program Object Resource (ap)

A KNX application program is a software that runs on a KNX device. The program is used to control and automate various functions within an installation.

The Application Program Functional Block (see [03] "KNX Information Model") contains global information about the internal application program, such as the Device Load State Machine.

Table 25 defines configuration resources for KNX IoT that a device MANDATORY ("M") supports.

Table 25 – Application Program resources and paths

Resource path	rt & Data Types	Format	Method	Support	Request/Response	Notes
ap	:fb.3	link-format	GET	**M**	Content-Format: application/link-format Payload: </a/lsm>;ct=60, </ap/pv>;rt=":dpa.3.13";ct=60	Application Program *Functional Block*.
ap/pv	:dpa.3.13 :dpt.programVersion	cbor (json)	GET PUT	**M**	Content-Format: application/cbor Payload: { 1: [1,2,3] }	The version of the Application Program.
a/lsm	NA	cbor	GET	**M**	Req: Content-Format: application/cbor Payload: { 3: 1 }	Function the Point Table state: 1 = "loaded"
a/lsm	NA	cbor	POST	**M**	Req: Content-Format: application/cbor Payload: { 2: 1 } Res: Content-Format: application/cbor Payload: { 3: 2 }	Change the Function Point Table state: 1 = "startloading" [1)] 2 = "loading"
					Req: Content-Format: application/cbor Payload: { 2: 2 } Res: Content-Format: application/cbor Payload: { 3: 1 }	Change the Function Point Table state: = "loadcomplete" 1 = "loaded"

[1)] Encoding of load commands and load states is reused from KNX Classic's LSM, see [09][10].

Resource path	rt & Data Types	Format	Method	Support	Request/Response	Notes
					Req: Content-Format: application/cbor Payload: { 2: 4 } Res: Content-Format: application/cbor Payload: { 3: 0 }	Change Function Point Table state and resets the resources "fp/g", "fp/r", "fp/p", and "p". 0 = "unloaded"

2.5.8.1 Device Load State Machine Command (a/lsm)

The Device Load State Machine controls the access to different resources and the runtime behavior of the application program.

If the Device Load State Machine is not loaded (e.g., unloaded), the following resources SHALL NOT be evaluated:

- Group Object Table (fp/g)
- Function Point Recipient Table (fp/r)
- Function Point Publisher Table (fp/p)
- Parameters (p)

That means runtime communication is impossible if the Device Load State Machine is not set to "loaded".

The transitions between the states SHOULD be less than 30 seconds. If a KNX IoT device cannot immediately respond after a state transition, for example, from "unloaded" to "loading" because the device restarts during the loading procedure, then the server SHALL return 2.04 CHANGED with the Max-Age option to indicate the number of seconds a MaC SHALL wait before sending the subsequent request.

The value of the Device Load State Machine SHALL be stored in non-volatile memory because the state has to be preserved after restart or power failure. Table 26 and Table 27 describe the MANDATORY ("M") and OPTIONAL ("O") states and events of the Device Load State Machine.

Table 26 – Device Load State Machine States

Load State [2)]	Support	Notes
unloaded (0)	**M**	No data is loaded. The associated resources have been reset to their default values.
Loaded (1)	**M**	Valid data is loaded. Only in this state the associated data SHALL be considered valid; in all other states, the data SHALL be considered invalid.

[2)] Encoding load states is reused from KNX Classic's LSM, see [10].

Load State [2]	Support	Notes
Loading (2)	**M**	Load process is active.
Unloading (4)	O	Unload process is active.
Loadcompleting (5)	O	Intermediate state between "loading" and "loaded".

Table 27 – Device Load State Machine Events

Load Events [3]	Support	Notes
nop (0)	O	No operation
startloading (1)	**M**	Request to start the loading of the loadable part
loadcomplete (2)	**M**	Request to complete the loading of the loadable part. Possible checksum SHALL be calculated, all data SHALL be declared valid, and the Device Load State Machine SHALL change to "loaded". If the transition from "loading" to "loaded" takes more than two seconds, e.g., by complex checksum calculation, the Load State shall first change to the intermediate state "loadcompleting" and change to "loaded" after the calculation is finished.
Unload (4)	**M**	Request to unload the loadable part.

The related resources SHALL be used immediately after the Device Load State Machine has reached the state "loaded".

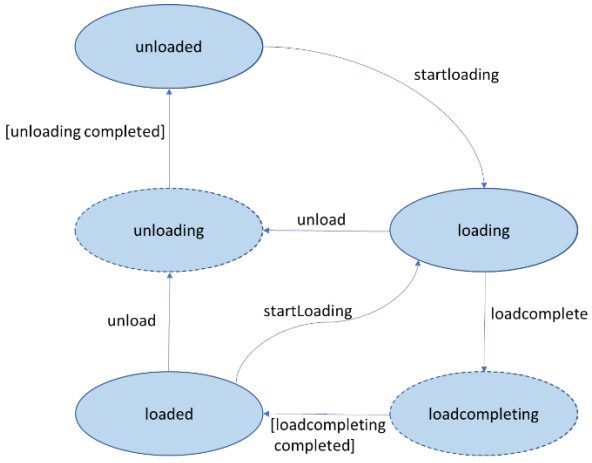


Figure 15 – Device Load State Machine

[3] Encoding load events is reused from KNX Classic's LSM, see [10].

2.5.8.2 Device Load State Machine Resource Object

Table 28 defines JSON keys and the CBOR mapping for CoAP request and response attributes of the "a/lsm" resource object:

Table 9 defines JSON keys and the CBOR mapping for CoAP request and response members of "/.well-known/knx". MANDATORY ("M") attributes SHALL be present in the message.

Table 28 – Device Load State Machine Resource Object

JSON Key	CBOR Key	CBOR Type	Support	Description
"cmd"	2	Text string	**M**	Service command. Enum: startloading, unload, loadcomplete
"status"	3	Text string	**M**	Current status. Enum: loaded, loading, unloaded, unloading, loadcompleting

2.5.9 S-Mode Messaging Resource (.knx)

2.5.9.1 Definition

A KNX IoT device SHALL send and receive messages on configured multicast addresses (S-Mode messaging endpoint) as described in clause 2.6.9, "S-Mode Group Communication".

The KNX IoT device (server) SHALL provide a resource ".knx" of resource type "rt=urn:knx:g.s" for unicast clients (device root path without /{base-path}).

If a client sends a valid unicast or multicast request (see 2.6.9) to ".knx" on a KNX IoT device (server), then the KNX IoT device SHALL check the Group Object Table if the corresponding "ga" is configured and write the "value" to the configured Point.

The KNX IoT device SHALL listen on configured multicast addresses (see 2.5.7.5).

A Message Broker such as a KNX IoT Router [12] MAY want to subscribe for S-Mode group notifications. Therefore, the resource ".knx" SHALL support subscriptions (CoAP observe), incl. "lt" and "non" query parameters according to clauses 2.5.9.3 and 2.5.9.4.

If a KNX IoT device (client) has successfully subscribed to ".knx" or sends a simple GET request, then the device SHALL return with a CoAP response code "2.05 Content"

A KNX IoT device SHALL provide the following resources for Publisher and Recipient communication. Table 29 defines S-Mode communication resources for KNX IoT that a device MANDATORY ("M") supports.

Table 29 – Resources for Publisher and Recipient Communication

Resource path	rt & Data Types	Format	Method	Support	Request/Response	Notes
.knx	:g.s	cbor	POST	**M**	Req: Content-Format: application/cbor Payload: (see example in clause 2.6.9)	The Publisher sends a notification to the KNX IoT device (Recipient).

Resource path	rt & Data Types	Format	Method	Support	Request/Response	Notes
.knx	:g.s	cbor	OBSERVE GET	**M**	Res: Content-Format: application/cbor Payload: (see example above and in clause 2.6.9)	Clients can subscribe to the S-Mode messaging from a device to get all S-Mode notifications from the device.

2.5.9.2 Group Notification Resource Object

To reduce message payload size, the following CBOR keys SHALL be used for Group Messages. The following fields are MANDATORY for S-Mode notification events:

- sia: The source *KNX Individual Address* SHALL be present in the message payload. If a Message Broker (KNX IoT Router) is between the Publisher and Recipient, then the Publisher *KNX Individual Address* SHALL be in the message payload. Leading zeros (e.g., multiple "0" in JSON) in the *KNX Individual Address* SHALL be omitted.
- s: The S-Mode message object SHALL contain a service type and a value for write and response service type.

Table 30 defines JSON keys and the CBOR mapping for CoAP S-Mode notification events. MANDATORY ("M") attributes SHALL be present in the message, and OPTIONAL ("O") may be present in the message but the attributes SHALL be supported on the device.

Table 30 – JSON keys for Group Object notification

JSON Key	CBOR Key	CBOR Type	Support	Description
"value"	1	see clause 2.5.13, "Datatype Mapping"	**M**	*Group Object* value
"sia"	4	Unsigned	**M**	Source ID (device *KNX Individual Address*).
"s"	5	map	**M**	S-Mode Group Message
"st"	6	text string	**M**	Service type code (write=w, read=r, ack/response=a) Enum: w, r, a
"ga"	7	unsigned	**M**	*Group Address*

2.5.9.3 Lifetime Query Parameter "lt"

The lifetime (seconds) of a subscription on ".knx" (S-Mode messages) SHALL be configured together with a CoAP observe request. If no lifetime query parameter of type unsigned integer is included in the initial CoAP observe or the requested lifetime is not supported by the server, then the server SHALL return a response code of "4.00 Bad Request".

A KNX IoT device SHALL support a minimum lifetime of 86400 seconds (24h).

2.5.9.4 Non-Confirmable Notification Query Parameter "non"

The CoAP observe request MAY contain a "non" query parameter of type boolean that indicates whether a resulting notification SHALL be sent over non-confirmable transport. The default setting for notification is confirmed ("non"=false) if "non" is absent in the CoAP observe request.

2.5.10 Functional Block Resource (f)

The *Functional Block* resource "f" provides a list of *Functional Blocks*.

The "f" resource MAY contain additional *Functional Blocks* beside the announced *Functional Blocks* in "/.well-known/core" (e.g., disabled functionality). However, the response SHALL contain at least the same *Functional Block* links (f/<fb-id-instance>) as in the "/.well-known/core" response.

The resource type ("rt") of a *Functional Block* SHALL contain the prefix ":fb." Followed by the *Functional Block* ID according to KNX standard Volume 7.

The "urn:knx" namespace identifier SHALL be omitted in the response to reduce payload.

The *Functional Block* resources "f/{fb-id-instance}" SHALL support the application/link-format containing links to the *Functional Block* Properties (p/{property-path}). This allows clients to discover Points with a subsequent request.

Table 31 defines *Functional Block* resources for KNX IoT that a device MANDATORY ("M") supports.

Table 31 – Functional Block resource

Resource path	rt & Data Types	Format	Method	Support	Request/Response	Notes
f	NA	application/link-format	GET	**M**	Res: Content-Format: application/link-format Payload: </f/rts>;rt=":fb.321";ct=40	The list contains all *Functional Blocks* of a device.
f/{fb-id-instance}	NA	application/link-format	GET	**M**	Res: Content-Format: application/link-format Payload: </p/rts/temproom>;rt="dpa:321.51";ct=50	The list contains all Properties of a *Functional Block*.

2.5.11 Parameter and Diagnostic Property Resource (p)

2.5.11.1 General Requirements

Parameter and diagnostic Properties are used for sensor, actuator, parameter, and diagnostic values, such as getting the current sensor value or setting a setpoint. Parameter and diagnostic Properties are addressed by URIs and can be directly accessed (read) and optionally manipulated (write). The following rules apply for CBOR as well as for JSON.

The {point-path} is a concatenation of `{base-path}`, the MANDATORY "p/" and a manufacturer specific `{property-path}`. The following examples are valid resource paths for Properties:

- *Functional Block* ID/instance and Property ID (see KNX standard Volume 7): p/321/1/51
- *Functional Block* and Property names (see KNX standard Volume 7): p/rts/temproom
- Simple numbers: p/1

[RFC8259] defines JSON as a serialized value. This means that a JSON value without brackets is also valid JSON (e.g., 25.5 or "TextString"). A GET response or, if supported, a CoAP observe notification [RFC8323] of simple Datapoint Types (see clause 2.5.13 "Datatype Mapping") SHALL be conveyed as simple JSON value without brackets to reduce the payload size. However, this specific rule SHALL NOT apply to CBOR. CBOR already has a compact format and SHALL be conveyed as a key/value pair.

CBOR and JSON Diagnostic Property read (w/ OSCORE) example for a value of a simple Datapoint Type.

REQ:
POST coap://{ipv6-unicast}/{point-path}
(OSCORE(code(GET), kid(<osc-id>)))

RES:
2.05 CONTENT (Content-Format: application/cbor (60))
Payload:
{ 1: 100.0 }

REQ:
POST coap://{ipv6-unicast}/{point-path}
(Accept: application/json (50), OSCORE(code(GET), kid(<osc-id>)))

RES:
2.05 CONTENT (Content-Format: application/json (50))
Payload:
100.0

The same rule applies if a client wants to change a Parameter Property; the JSON payload SHALL convey the value without brackets.

CBOR and JSON Parameter Property write (w/ OSCORE) example for a value of a simple Datapoint Type.

REQ:
POST coap://{ipv6-unicast}/{point-path}
(Content-Format: application/cbor (60), OSCORE(code(PUT), kid(<osc-id>)))
Payload:
{ 1: 20.0 }

REQ:
POST coap://{ipv6-unicast}/{point-path}
(Content-Format: application/json (50), OSCORE(code(PUT), kid(<osc-id>)))
Payload:
20.0

Complex Datapoint Types SHALL be conveyed without a "value" key (only JSON) as a simple array according to clause 2.5.13, "Datatype Mapping", to reduce the payload size.

CBOR and JSON Diagnostic Property read (w/ OSCORE) example for a value of a complex Datapoint Type.

REQ:
POST coap://{ipv6-unicast}/{point-path}
(OSCORE(code(GET), kid(<osc-id>)))

RES:
2.05 CONTENT (Content-Format: application/cbor (60))
Payload:
{ 1: [13.0, 18.0, 20.0, 21.0] }

REQ:
POST coap://{ipv6-unicast}/{point-path}
(Accept: application/json (50), OSCORE(code(GET), kid(<osc-id>)))

RES:
2.05 CONTENT (Content-Format: application/json (50))
Payload:
[13.0, 18.0, 20.0, 21.0]

However, in some cases, Parameter or Diagnostic Property values are conveyed with additional data. In that case, the JSON value SHALL be identified by the key "value". For example, if the value has additional status information (see clause 2.5.11.5 "Status and Command Resource Object (Z8)") or the value is conveyed with other metadata (see clause 2.5.11.3, "Metadata Query Parameter "m"). Therefore, a JSON client SHOULD consistently implement both message types (only value and as key/value pair).

Table 32 defines configuration resources for KNX IoT that a device MANDATORY ("M") supports.

Table 32 – Parameter and Diagnostic Properties

Resource path	rt & Data Types	Format	Method	Support	Request/Response	Notes
p	NA	application/link-format	GET	O	Res: Content-Format: application/link-format Payload: </p/rts/temproom>;rt=":dpa.321.51";ct=60	The list contains all datapoints of a device.

Resource path	rt & Data Types	Format	Method	Support	Request/Response	Notes
p	NA	cbor, (json)	POST	M	Req: Content-Format: application/json Payload: [{ "href ":"<point-path>", "value":"abc " }, { "href":"<point-path>", "value":21.0 "min":20.0, "max":25.0 }, …]	Writes a set of parameters to a KNX IoT device. Each collection item SHALL contain an "href" resource identifier.

2.5.11.2 Point Value Update Notification

Notifications to Subscriber devices SHALL be based upon the criteria contained in application configurations (e.g., lighting or HVAC) and are activated with the transmission configuration "cflag" (transmit). A notification SHALL be sent when the heartbeat time has elapsed since the previous message or the configured change of value (CoV) has exceeded the same value. Heartbeat time, MinRepTime, or change of value (CoV) are optional metadata members; see clause 2.5.11.3, "Metadata Query Parameter "m" for more details. However, a notification SHALL NOT be sent before the minimum repetition time (MinRepTime) has elapsed.

All Subscription notifications with a configured heartbeat time SHALL contain a Max-Age CoAP option containing the remaining time.

Hence, *Group Objects* and subscriptions supporting event notifications require the specification of the following metadata parameters:

- CoV
- MinRepTime
- Heatbeat

2.5.11.3 Metadata Query Parameter "m"

Metadata members are Point attributes used to annotate, tag, and describe *Group Objects*, Parameters, and Diagnostic Properties. Every Point has a set of metadata members. Metadata members are modeled as key/value pairs and addressable via the {point-path} and query parameters as specified and shown below.

The following resources SHALL support metadata but other resources MAY support metadata members as well:

- /{base-path}/p/{property-path}

To retrieve the value of {metadata-member}, a KNX IoT device SHALL accept a GET request to the URI {point-path}?m={metadata-member}&m={metadata-member}. If at least one of the requested metadata {metadata-member} is supported, then the device SHALL return a response with CoAP response code "2.05 Content". The response payload SHALL contain the {metadata-member} as key, and the {metadata-value} of the specified metadata as value but missing or not supported metadata member fields (e.g., unit) SHALL be omitted in the response.

If none of the requested metadata {metadata-member} is supported, the device SHALL return a response with CoAP response code "4.04 Not Found" with an empty payload.

Suppose the client requests a specific {metadata-member} (e.g., m=unit), but the metadata member field is not supported on this datapoint; then the device SHALL return a response with CoAP response code "5.01 Not Implemented".

CBOR Metadata read example for selected metadata members.

REQ:
GET coap://{ipv6-unicast}/{point-path}?m=min&m=max

RES:
2.05 CONTENT (Content-Format: application/cbor (60))
Payload:
{ "min": 0.0, "max": 50.0 }

The metadata query parameter is used with GET to retrieve a subset of metadata items for the selected resource. The parameter has the following form: m={metadata-member} for which {metadata-member} can be either a single metadata identifier or wildcard ("m" or "m=*") to select all existing metadata.

JSON Metadata read example for all metadata members.

REQ:
GET coap://{ipv6-unicast}/{point-path}?m
(Accept: application/json (50))

RES:
2.05 CONTENT (Content-Format: application/json (50))
Payload:

```
{
 "id":"knx://sn.<serialnumber>/p/rts/temproom",
 "rt": [":dpa.321.51"],
 "dpt":":dpt.valueTemp",
 "value": 21.0,
 "if": [":if.o", ":if.d"],
 "unit": "DEG_C",
 "min": 0.0,
 "max": 50.0,
 "mrt": 2,
 "cov": 0.5,
 "hbt": 900,
 "sns": true,
 "ga": [1234]
}
```

2.5.11.4 Metadata Resource Object

KNX IoT defines a standardized basic set of metadata that MAY be applied to Points (see clause 5.1.3). Standardized KNX IoT metadata resource names (key) do not contain a namespace.

All Point metadata resources SHALL support the CBOR format (see clause 2.2.4 "Content-Format"). Table 33 defines the metadata members for KNX IoT that the server MANDATORY ("M") or OPTIONAL ("O") supports. (PUT) on metadata is OPTIONAL since these values are often preconfigured (e.g., with MaC).

Table 33 – KNX IoT common metadata

JSON Key	CBOR Key	CBOR Type	Method	Support	Notes
"id"	9	text string	GET	**M**	ID: An identifier for a data item that is not dependent on the data item's location. It SHALL be permanent and SHALL move with the data if the data changes its location. The ID SHOULD use the *KNX Serial Number* LRI format defined in 2.6.1.4 "Linkage and Resolution" (e.g., knx://sn.1caffe1234/p/rts/temproom)
"href"	11	text string	GET	**M**	Reference to the Point, which represents the data of this Point. The "href" SHALL be used as a Point identifier for writing a Point collection but can be omitted in a metadata query response.
"value"	1	see DPT	GET (PUT)	**M**	Current datapoint value.
"rt"	"rt"	text string	GET	**M**[a]	Type: The specific type name of the data item is equivalent to a resource type (rt). A Point with a specific type can be discovered via /.well-known/core?rt={type} (see clause 2.6.1.3.6.4 "Resource Type "rt"").
"if"	"if"	text string	GET	**M**[a]	Interface: An interface provides a view for organizing Points for discovery (see clause 2.5.3 "Interface Types (if)") and for interactions, such as access rights. Points can be discovered via /.well-known/core?if={if}. Therefore, the values SHALL contain the relative URN format (see clause 1.4.5) of the interface type (e.g., ":if.sec").
"desc"	"desc"	text string	GET (PUT)	O	Description: A localizable description of the data item.
"dpt"	"dpt"	text string	GET	**M**[a]	KNX Datapoint Type (see clause 2.5.11.4.1.1)

JSON Key	CBOR Key	CBOR Type	Method	Support	Notes
"unit"	"unit"	text string	GET	O	Unit: The engineering unit of the data item. KNX units are defined by QUDT (http://qudt.org/vocab). For example, an air temperature unit (°C) is encoded as follows: DEG_C.
"min"	"min"	float	GET (PUT)	O	Min: The minimum value allowed for the data item (Point).
"max"	"max"	float	GET (PUT)	O	Max: The maximum value allowed for the data item (Point).
"mrt"	"mrt"	integer	GET (PUT)	O	MinRepTime: Minimum delay in seconds until the next message is sent.
"cov"	"cov"	integer	GET (PUT)	O	Change of Value: The Point sends a notification message if the point's value change has exceeded this Change of Value (CoV) threshold.
"hbt"	"hbt"	integer	GET (PUT)	O	Heartbeat: A Point sends a periodic value update to indicate normal operation or to synchronize other devices.
"sns"	"sns"	integer	GET (PUT)	O	Sequence Number Synchronization: Sequence number synchronization is disabled if false or the metadata member is absent. If configured (true), then a read/write request to the Point value triggers message counter synchronization (see also clause 3.6.4 "Message Replay Protection").
"ga"	"ga"	Array [unsigned]	GET	**(M)**	*Group Address*: Configured *Group Addresses* for S-Mode communication (<group-address>). A Point with a group-address can also be discovered via /.well-known/core?d=urn:knx:g.s.{group-address}.

[a] The value SHALL be an empty string ("") if the resource definition states NA.

Metadata members on a KNX IoT device MAY be writable. To set the value of {metadata-member}, a KNX IoT device MAY accept a PUT request to the URI {point-path} containing one {metadata-member} and its new value in the payload. A single PUT request payload SHALL contain only one writable {metadata-member} as "value": <metadata-value> key/value pair.

CBOR Metadata write example.

REQ:
POST coap://{ipv6-unicast}/{point-path}?m=min
(Content-Format: application/cbor (60), OSCORE(code(PUT), kid(<osc-id>)))
Payload:
{ 1: 0.0 }

RES:
2.04 CHANGED

2.5.11.4.1.1 Datapoint Type "dpt"

The KNX datapoint type is a combination of a data type and a dimension. The "dpt" attribute is an opaque string indicating a specific KNX datapoint type. The KNX Classic DPT names from KNX standard Chapter 3/7/2, "Datapoint Types" SHALL be used. The following conversion rules SHALL be applied on KNX Classic DPT names for KNX datapoint types (see also DPT names in [03] "KNX Information Model"):

- "DPT_" is removed.
- "_" is removed.
- square brackets [] are replaced with "a" and "a_".
- "/" is replaced with "p".
- numbers are kept.
- terms expressing a proper name, or a specific abbreviation shall be converted to lowercase; this rule is rather a list than an exact (regular) expression; see the example below:
  - HVAC to hvac, HPM to hpm, RGB to rgb, ASCII to ascii, DALI to dali, Mbus to mbus
- terms expressing a (combined) unit shall not be changed; see the example below:
  - kVARh, MWh, mm, 100Msec
- the resulting string is prefixed with "urn:knx:dpt." in the case of a standard KNX DPT. The leading "urn:knx" prefix SHALL be omitted on the "dpt" metadata of a Point (e,g., ":dpt.valueTemp").

2.5.11.5 Status and Command Resource Object (Z8)

In CoAP, the Z8 status/command feature (see KNX Specifications Chapter 3/7/2) is an optional feature of a Point. If a value is out-of-service, overridden, or in a fault state, this information is provided with the value as additional information. Consequently, all Z8 data types (e.g., V16Z8, U8Z8, etc.) SHOULD be mapped to simple data types without Z8.

For Points supporting the Z8 feature, a KNX IoT device SHALL deliver Z8 additional status information accompanied by the value only if the status deviates from "normal". A client SHALL handle values accompanied with additional Z8 status information, especially when the value is not present but only the Z8 status information, for example, in case the value is out-of-service.

The status/command extension Z8 has a dual representation, i.e., the representation is different depending on the role of status (response, info report paradigm) or command (write paradigm).

1279 **Table 34 – Status and Command JSON/CBOR keys**

JSON Key	CBOR Key	CBOR Type	Description
"cmd"	2	unsigned	KNX Z8 command is represented by an 8-bit enumeration => CBOR unsigned. The following commands MAY are supported on a Point: - override: Temporary override of a sensor or actuator value. - release: Undo "override", leads to normal operation of the Point using the actual value. - setOSV: Disable functionality of a Point such as a configuration parameter is void (function disabled) or a sensor is disabled. - resetOSV: Undo "setOSV", leads to normal operation of the Point using the actual value.
"overridden"	111 ("o")	bool	True = Value is currently overridden
"osv"	"osv"	bool	True = No valid value available
"fault"	102 ("f")	text string	Error incl. fault information

1280

1281 In the presence of Z8 values, the payload is represented as a map, and the Z8 is embedded in the payload
1282 under the "status" resp. "cmd" key. JSON examples:

1283 Value status = normal: 100

1284 Value status = overridden: { "value": 100, "overridden": true }

1285 Value status = out-of-service: { "osv": true }

1286 Value status = fault: { "fault": "general fault" }

1287 Value command = override/release: { "value": 50, "cmd": "override" }

1288 Value command = setOSV/resetOSV: { "cmd": "setOSV" }

Command Example.

REQ:
POST coap://{ipv6-unicast}/{point-path}
(Content-Format: application/json (50), Accept: application/json (50), OSCORE(code(POST), kid(<osc-id>)))
Payload:
{ "value": 50, "cmd": "override" }

RES:
2.05 CONTENT (Content-Format: application/json (50))
Payload:
{ "value": 50, "overridden": true }

1289

2.5.11.6 Lifetime Query Parameter "lt"

The lifetime (seconds) of a subscription on parameter and diagnostic Properties SHALL be configured together with a CoAP observe request. If no lifetime query parameter of type unsigned integer is included in the initial CoAP observe or the requested lifetime is not supported by the server, then the server SHALL return a response code of "4.00 Bad Request".

A KNX IoT device SHALL support a minimum lifetime of 86400 seconds (24h).

2.5.12 Subscription Resource (sub)

2.5.12.1 Definition

A KNX IoT device MAY support subscriptions on predefined Points, described in clause 2.6.10, "Point Publish/Subscribe".

Table 35 specifies the MANDATORY ("M") subscription resources (sub) and the corresponding resource path names that SHALL be used if the KNX IoT device supports subscriptions.

Table 35 – Mandatory and optional subscription resources

Resource path	rt & Data Types (rt)	Format	Method	Support	Request/Response	Notes
sub	NA	NA	DELETE	**M**	Res: 2.02 DELETED	Delete all device CoAP observe subscriptions.

2.5.13 Datatype Mapping

2.5.13.1 Introduction and general requirements

The following tables illustrate the representation of KNX datatypes in CBOR and JSON.

In some cases, as possible with CBOR, variable-length encoding will provide a shorter notation, e.g., unsigned numbers < 24 will be encoded in a single byte in CBOR, even if KNX Classic datatype representation is a 32-bit unsigned number.

Three categories of datatypes are discerned:

1. KNX simple datatypes: direct mapping to a simple CBOR/JSON datatype exists.
2. KNX complex datatypes: two or more simple CBOR/JSON datatypes are needed - those will be aggregated in an array.
3. KNX Z8 data types: has different representations depending on the service context used (status or command).

NOTE 3 Datatype modeling is not limited to KNX datatypes known from KNX Classic. I.e., further data types or data object structures may be defined that are used by features that may apply only to KNX IoT. An example of doing so may be modeling data for trend logs.

2.5.13.2 Simple Datatypes

Table 36 – Simple datatypes and CBOR representation

KNX data type	CBOR Type	JSON Type [a]	Notes
B1	bool	boolean	
A8	text string	string	UTF-8 representation, ASCII is identical, ISO-8859-1 needs conversion to UTF8

KNX data type	CBOR Type	JSON Type [a]	Notes
U8	unsigned	integer	
V8	negative / unsigned	integer	
U16	unsigned	integer	
V16	negative / unsigned	integer	GET and PUT to Parameter, and Diagnostic Properties containing a temperature value are mapped to CBOR float and JSON Number.
F16	float	number	F16 is incompatible with CBOR 16-bit float, thus using 32-bit float. IEEE754 encodes with 1-bit sign, 5-bit exponent with bias 15 [-14,15] and 10-bit significant - with 11-bit precision under the assumption of a leading 1 if the exponent is not 0 (see https://en.wikipedia.org/wiki/Half-precision_floating-point_format). KNX F16 encodes 12-bit of mantissa in 2 complement representation with a 4-bit exponent [0-15] and an assumed precision of 0.01. this leads to a range from [671088.64, 670670,96]. The encoding is different and leads to different ranges.
U32	unsigned	integer	
V32	negative / unsigned	integer	
F32	float	number	CBOR has IEEE754 style floats - these match with the KNX 32-bit float definition.
A112	text string	string	Conversion to UTF-8 representation needed, ASCII is valid, ISO-8859-1 needs conversion to UTF8. Max length is 14 characters - no trailing 00 are encoded.
N2, N3, N5, N6, N8	unsigned	integer	If the numeric value is less than 24, 1 byte is sufficient.
A[n]	text string	string	Conversion to UTF-8 representation is needed if not UTF8 already. The length is dynamic - no trailing 00 as EOL are encoded.
V64	negative / unsigned	integer	

[a] JSON can be basic types (integer, boolean, …), objects, or arrays. Objects "{}" are unordered and need to be populated with key:value pairs.
Note: RFC8259 specifies *number* as the only numeric value type. However, JSON schema defines *integer* as a separate value type.

2.5.13.3 Complex Datatypes

KNX complex datatypes are modeled as an array of aggregated simple CBOR/JSON datatypes.

KNX-structured DPTs are mapped to CBOR/JSON datatypes by the following rules:

- represent singular fields of the KNX datatype (DPT) with their simple datatype (acc. 2.5.13.2 "Simple Datatypes") and concatenate them to CBOR/JSON arrays
- use next larger, simple data types for length-optimized fields of KNX data types, to map them to CBOR/JSON simple data types (e.g., U3 → U8 → unsigned/number)
- reserved KNX data type / DPT fields SHALL be omitted - in most cases, the modeling by the next larger data type allows for extension

Table 37 lists complex datatypes and the defined CBOR representation.

1333 **Table 37 – Complex datatypes and CBOR representation**

KNX data type	CBOR Type	JSON Type [a]	Notes
B2	[bool, bool]	[bool, bool]	
B1U3	[bool, unsigned]	[bool, number]	
B5N3	[[bool*], unsigned]	[[bool*], integer]	6.020 DPT_Status_Mode3
N3U5r2U6r2U6	[unsigned, unsigned, unsigned, unsigned]	[integer, integer, integer, integer]	10.001 DPT_TimeOfDay
r3U5r4U4r1U7	[unsigned, unsigned, unsigned]	[integer, integer, integer]	11.001 DPT_Date
U4U4U4U4U4U4B4N4	[unsigned, unsigned, unsigned, unsigned, unsigned, unsigned, [bool*4], unsigned]	[integer, integer, integer, integer, integer, integer, [bool*4], integer]	15.000 DPT_Access_Data
r2U6	unsigned	integer	17.001 DPT_SceneNumber
B1r1U6	[bool, unsigned]	[bool, integer]	18.001 DPT_SceneControl
U8[r4U4][r3U5][U3U5]-[r2U6][r2U6]B16	[unsigned*7, [bool*9]]	[integer*7, [bool*9]]	19.001 DPT_DateTime: the ending B16 of the DPT is encoded from left to right and ends with seven (7) padding "zeros", i.e., only the nine (9) leading bits provide relevant information
B8	[bool*]	[bool*]	
B16	[bool*]	[bool*]	
U4U4	[unsigned, unsigned]	[integer, integer]	
r1b1U6	[bool, unsigned]	[bool, integer]	
B32	[bool*]	[bool*]	
B24	[bool*]	[bool*]	
U16N8	[unsigned, unsigned]	[integer, integer]	
U8B8	[unsigned, [bool*]]	[integer, [bool*]]	
V16B8	[negative / unsigned, [bool*]]	[integer, [bool*]]	
V16B16	[negative / unsigned, [bool*]]	[integer, [bool*]]	
U8N8	[unsigned, unsigned]	[integer, integer]	

KNX data type	CBOR Type	JSON Type [a]	Notes
V16V16V16	[negative / unsigned, negative / unsigned, negative / unsigned]	[integer*3]	
V16V16V16V16	[negative / unsigned, negative / unsigned, negative / unsigned, negative / unsigned]	[integer*4]	
V16U8B8	[negative / unsigned, unsigned, [bool*]]	[integer, integer, [bool*]]	
V16U8B16	[negative / unsigned, unsigned, [bool*]]	[integer, integer, [bool*]]	
U16U8N8B8	[unsigned, unsigned, unsigned, [bool*]]	[integer, integer, integer, [bool*]]	
U5U5U6	[unsigned, unsigned, unsigned]	[integer*3]	
U8N8N8N8B8B8	[unsigned, unsigned, unsigned, unsigned, [bool*], [bool*]]	[integer, integer, integer, integer, [bool*], [bool*]]	
U16V16	[unsigned, negative / unsigned]	[integer, integer]	
N16U32	[unsigned, unsigned]	[integer, integer]	
F16F16F16	[float, float, float]	[number, number, number]	
V8N8N8	[negative / unsigned, unsigned, unsigned]	[integer, integer, integer]	
V16V16N8N8	[negative / unsigned, negative / unsigned, unsigned, unsigned]	[integer, integer, integer, integer]	
U16U8	[unsigned, unsigned]	[integer, integer]	

KNX data type	CBOR Type	JSON Type [a]	Notes
U16U32U8N8	[unsigned, unsigned, unsigned, unsigned]	[integer, integer, integer, integer]	
A8A8A8A8	text string	string	
U8U8U8	[unsigned, unsigned, unsigned]	[integer, integer, integer]	
A8A8	text string	string	

[a] JSON can be basic types, objects (maps), or arrays. Maps "{}" are unordered and have to be populated with key:value pairs.
Note: RFC8259 specifies *number* as the only numeric value type. However, JSON schema defines *integer* as separate value types.

1334

1335 #### 2.5.13.4 KNX IoT Datatype Extensions

1336 Table 38 lists datatypes that do not exist in KNX Classic and have been introduced for KNX IoT.

1337 **Table 38 – Datatype Extension CBOR representation**

KNX Data Type	CBOR Type	JSON Type	Name	Notes
O[16]	byte string	string	DPT_ipv6	The IPv6 address data type octet string SHALL correspond to the total 16 octets according to [RFC4291] in network byte order. Example: IPv6 Address: 2001:db8:3::6cd9:8ad2:8e88:1f68 -> Octet string: h'20010db8000300006cd98ad28e881f68'
U16U16U8	[unsigned, unsigned, unsigned]	[integer, integer, integer]	DPT_programVersion	Program Version SHALL be formatted as follows (see also [10]): [<Manufacturer ID>, <Device Type>, <Application Program Version>]

1338

2.6 Runtime Interworking

2.6.1 Discovery

2.6.1.1 Introduction

A KNX IoT device supports different discovery features that a client can use to learn about a KNX IoT device, for example, as a part of the device bootstrapping and configuration process (see clause 2.4). KNX IoT defines a DNS-based service discovery (DNS-SD) for IP address resolution in combination with a semantic resource discovery (incl. semantic annotations) based on CoAP. The device discovery is used within KNX IoT for the following purposes:

- Discover new device: MaC discovers new devices on the network. The pre-requisite is that devices have an IP address but no security configuration or *KNX Individual Address*.
- Discover commissioned device: A client discovers a commissioned KNX IoT device with at least security and a *KNX Individual Address* configured.
- Device-initiated MaC notification: The device initiates the MaC to start the commissioning procedure (*Programming Mode* enabled).

NOTE 4 This version of the KNX IoT Point API specification does not define the Device initiated commissioning.

2.6.1.2 Device Discovery with DNS-SD

2.6.1.2.1 Motivation

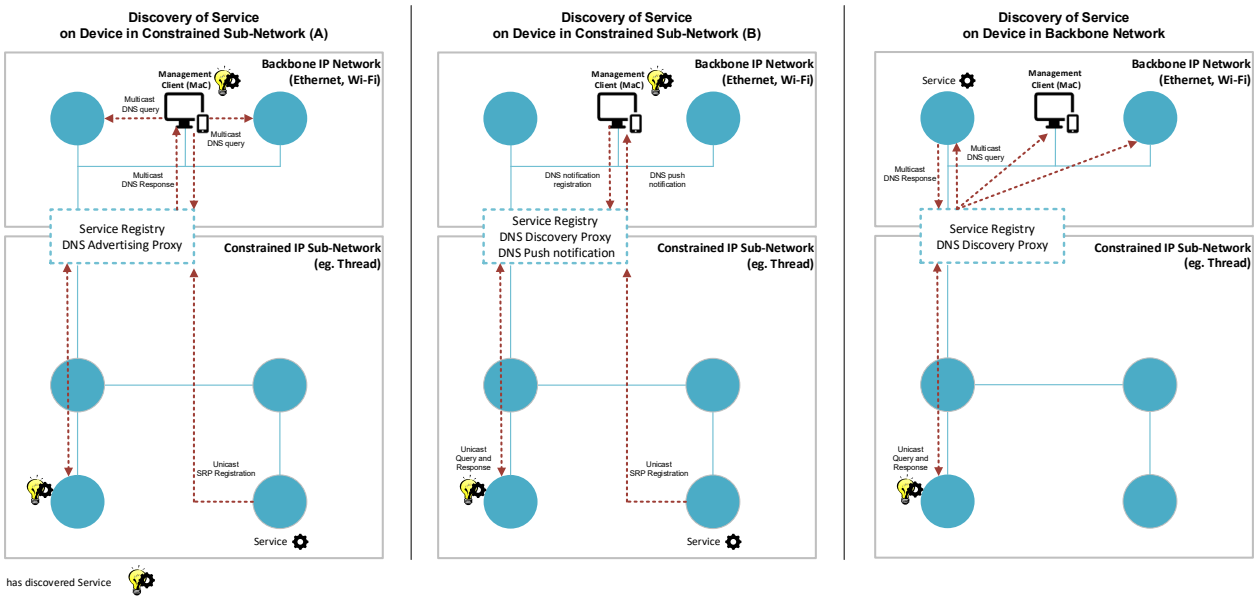
In constrained networks (e.g., Thread), multicast SHOULD be avoided since this can lead to uncontrolled responses or congestion issues in the constrained network. In addition, multicast messages need cache memory on routers with sleepy end nodes, which may lead to loss of packets, for example, if the cache is full.

Therefore, a KNX IoT device SHALL provide DNS Service Discovery (DNS-SD) [RFC6763] services to be discovered. Devices in a subsystem, for example, in a constrained network (e.g., Thread), MAY support this functionality by a DNS-SD Discovery Proxy [RFC8766]. Hence, a KNX IoT client can assume that (sleepy and non-sleepy) devices are always discoverable.

A client MAY perform a short-lived query using unicast DNS over UDP to the DNS-SD Discovery Proxy. If supported by the DNS-SD Discovery Proxy, clients MAY use DNS Push Notifications [RFC8765] with DNS Stateful Operations [RFC 8490] for long-lived queries to reduce expensive polling.

The KNX IoT device itself or a device discovery proxy SHALL advertise the device endpoint (service name) through a multicast Domain Name System (mDNS) [RFC6762]. KNX IoT devices in a subsystem with a DNS-SD Discovery Proxy, for example, a Thread network, SHALL support the DNS-SD Service Registration Protocol (SRP) [https://tools.ietf.org/html/draft-ietf-dnssd-srp] and implement an SRP-client. The SRP client registers the KNX IoT service name at the DNS-SD discovery proxy. A device discovers or becomes discoverable via DNS-SD discovery proxy in the local multicast scope and beyond without using multicast in the constrained network. Figure 16 illustrates such behavior for use cases (A) and (B).

The MaC MAY use mDNS in networks not providing any additional DNS services or for the initial discovery of better performing DNS services (see Figure 16, use case (A)). A MaC in larger networks with, e.g., a discovery proxy SHOULD use DNS Push Notifications [RFC8765] with DNS Stateful Operations [RFC8490] for improved user experience (see Figure 16, use case (B)). Also, constrained KNX IoT devices needing to discover devices MAY implement DNS Push for interaction with the DNS Discovery Proxy (see Figure 16, use case (C)).



(A) Service Registry Protocol (SRP) and DNS Advertising Proxy (ADVPROXY) for legacy mDNS support
(B) Service Registry Protocol, DNS Discovery Proxy, and DNS Push.
(C) Examples of Constrained KNX IoT devices discovering a service in the backbone IP network.

Figure 16 - Bidirectional DNS Service Discovery for KNX IoT device discovery

As described in [RFC6762], mDNS SHALL use UDP port 5353 with the multicast IP address FF02::FB for IPv6.

For standardizing the DNS-based discovery, a KNX IoT device SHALL adhere to the following rules:

1. The KNX IoT device or a DNS-SD proxy (e.g., Thread Border Router) SHALL implement an mDNS client providing its IP address in a resource record of type AAAA for IPv6 [RFC3596] and MAY provide the IP address in a resource record type A for IPv4 [RFC1035] if IPv4 is supported on the device (e.g., KNX IoT Router).
2. The KNX IoT device or the DNS-SD proxy SHALL support direct communication over at least one port. The name and port are propagated using the mDNS-SD protocol as defined in [RFC6763].
3. The DNS-SD service type [RFC6335] is "_knx._udp". The communication port can be selected without limitations. The service domain is ".local" for mDNS (default) or as configured by the DNS Discovery Proxy for unicast DNS from corresponding subnets [RFC8766, clause 5.1]. The target host name, required in addition to the instance name (see clause 2.6.1.2.2 "DNS-SD Services"), is the *KNX Serial Number* encoded as fixed-length UTF-8 text in lowercase letters, e.g., "001caffe1234.knx.local". If the device performs address randomization for privacy protection, the target host name updates accordingly.
4. Communication between different subnets is considered to be part of the infrastructure installation for IP communication. mDNS can offer proxy information that offers access to other subnets IF the corresponding routes have been added AND the DHCP server has offered the corresponding gateway.

NOTE 5 RFC6335: "The service name "_knx._udp" will be registered with IANA": http://www.iana.org/assignments/port-numbers

2.6.1.2.2 DNS-SD Services

The following standard DNS resource records represent a KNX IoT device in DNS-SD. KNX IoT uses standard DNS-SD Service Discovery according to [RFC6763], and no modifications are required. Irrelevant parts of the corresponding DNS resource records are omitted here for clarity.

The "Service" indicates the KNX IoT device DNS-SD service type or subtype. The "Domain" (e.g., local) is the DNS domain where the service type is present. The "Domain" is typically used to limit the resource records returned in a DNS query. The domain is independent of the domain name of the host of the DNS-SD service instance. The domains to scan for KNX devices MAY be discovered through reading DNS-SD defined resource records. The hostname identifies the service and is a readable name and MAY be determined locally but SHALL be unique among all instances of this service as defined in clause 2.6.1.2.3. For allowed characters, see [RFC6763], clause 4.1.1.

A KNX IoT device SHALL advertise DNS AAAA records since KNX IoT uses IPv6 communication. A KNX IoT device SHALL publish all AAAA records for all IPv6 addresses that are accepted for commissioning or runtime communication.

A KNX IoT device MAY advertise additional TXT records (see clause 2.6.1.2.4).

As defined in [RFC1035], PTR Resource Records are used for enumerating the KNX IoT device DNS-SD service instances. A KNX IoT device SHALL advertise the following service subtypes with the *KNX Serial Number* as the service name (ASCII hexadecimal string in lowercase letters) for the corresponding service instance:

- _{serialnumber}: Contains the *KNX Serial Number* as ASCII hexadecimal string in lowercase letters.
- _ia{installationid}-{ia}: MaC configured *KNX Individual Address* concatenated with the KNX Installation ID as ASCII hexadecimal string in lowercase letters (see clause 2.6.1.2.3).
- _pm: The device advertises this subtype only in *Programming Mode* (see clause 2.5.3.9).
- _m{manufacturer-specific-service}: Manufacturer-specific subtypes concatenated with a manufacturer-defined service name.

EXAMPLE 01 The example below shows a commissioned KNX IoT device in programming mode:

001caffe1234._knx._udp.local.	PTR 001caffe1234._knx._udp.local.
_ia33a3-20a._sub._knx._udp.local.	PTR 001caffe1234._knx._udp.local.
_pm._sub._knx._udp.local.	PTR 001caffe1234._knx._udp.local.
_mmyapi._sub._knx._udp.local.	PTR 001caffe1234._knx._udp.local.
001caffe1234._knx._udp.local.	SRV 0 0 5683 {hostname}.local.
{hostname}.local.	AAAA fe80::f244:126f:6772:4a60

2.6.1.2.3 Commissioning and Operational Discovery

The naming convention for the DNS-SD {hostname} of a KNX IoT device in default configuration state SHALL be in the format knx-{serialnumber}.local. The *KNX Serial Number* SHALL be a valid DNS name as an ASCII hexadecimal string in lowercase letters, e.g., "knx-001caffe1234.local". DNS-SD will detect a name conflict if the hostname is not unique. The device has to generate an alternative instance name according to [RFC6762] and the Service Registration Protocol [SRP]. It is RECOMMENDED to increment a number separated by a dash at the end of the hostname (e.g., knx-001caffe1234-1.local) in case of a name conflict. This name convention allows the discovery of a new device without the additional configuration of standard DNS servers. The hostname MAY change later in a local deployment.

A KNX IoT device in *Programming Mode* (see clause 2.5.3.9) that is waiting for commissioning SHALL advertise "_pm" as a subtype of the service type to allow for filtered responses (_pm._sub._knx._udp.local). A manufacturer-specific string MAY be provided as an additional subtype.

The following example shows a command-line example to create an mDNS advertisement using the avahi-utils Linux package and the subsequent discovery and IP address resolution. The device has a factory default configuration setting for the *KNX Individual Address* (e.g., 0).

Discovery example of a device in *Programming Mode*.

```
avahi-publish --host=knx-001caffe1234.local -s --subtype=_001caffe1234._sub._knx._udp --
subtype=_pm._sub._knx._udp --subtype=_ia-0._sub._knx._udp 001caffe1234 _knx._udp 5683


avahi-browse -rt _pm._sub._knx._udp
= IPv6 001caffe1234
   hostname = [knx-001caffe1234.local]
   address = [fd11:2222:33a3:0001:6cd9:8ad2:8e88:1f68]
   port = [5683]
```

For operational discovery, after the MaC has configured a *KNX Individual Address* and operational credentials (LDevID or OSCORE) etc., the DNS-SD service type of the operational KNX IoT device SHALL comprise the KNX Installation ID and the configured *KNX Individual Address* separated by a hyphen ("-"), encoded as ASCII hexadecimal string of 12 respectively 16 characters, e.g., _ia33a3-20a.knx.local, the target host name remains unchanged. Leading zeros in the KNX Installation ID and *KNX Individual Address* SHALL be omitted (e.g., 0x00000033a3 = _ia33a3-20a.knx.local).

During normal operation, a KNX IoT device SHALL advertise both the *KNX Serial Number* and the *KNX Individual Address* service type.

The following command-line example shows a DNS-SD advertisement using the avahi-utils Linux package and the subsequent discovery and IP address resolving.

Discovery example with *KNX Serial Number*.

```
avahi-publish --host=knx-001caffe1234.local -s --subtype=_001caffe1234._sub._knx._udp 001caffe1234
_knx._udp 5683

avahi-browse -r _001caffe1234._sub._knx._udp

= IPv6 001caffe1234._sub._knx._udp
   hostname = [knx-001caffe1234.local]
   address = [fd11:2222:33a3:0001:6cd9:8ad2:8e88:1f68]
   port = [5683]
```

2.6.1.2.4 TXT Records

TXT records are multi-purpose text-based key/value pairs that MAY be contained in a DNS-SD TXT record. Client devices SHALL silently ignore TXT records if they do not use the information.

2.6.1.2.4.1 Sleep Period Record "SP"

A sleepy KNX IoT device SHALL advertise the "SP" TXT record. The "SP" TXT record announces how long in milliseconds a sleepy KNX IoT device maybe not reachable. The "SP" value SHALL NOT exceed 1 hour (3’600’000 milliseconds).

EXAMPLE 02 The example below shows a sleepy KNX IoT device with 3 min. sleep periods:

```
001caffe1234._knx._udp.local.        PTR 001caffe1234._knx._udp.local.
_ia33a3-20a._sub._knx._udp.local.    PTR 001caffe1234._knx._udp.local.
001caffe1234._knx._udp.local.        TXT "SP=3000"
001caffe1234._knx._udp.local.        SRV 0 0 5683 {hostname}.local.
{hostname}.local.                    AAAA fe80::f244:126f:6772:4a60
```

2.6.1.3 Resource Discovery with CoAP

2.6.1.3.1 Introduction

Multicast resource discovery in CoAP (CoRE) is accomplished by using the "/.well-known/core" resource URI that returns a list of links about resources hosted by that server that matches filter attributes.

Multicast Resource Discovery is useful if a client needs to locate a resource within a limited scope, and that scope supports IP multicast. To limit the number and size of responses, a query string with known attributes (e.g., rt) is RECOMMENDED. Typically, a resource would be discovered based on its Resource Type (rt) along with possible application-specific attributes.

The Resource Type "rt" attribute is an opaque string used to assign an application-specific semantic type to a resource. In the case of a room temperature sensor (RTS) resource, this could be, e.g., an application-specific semantic type like URI referencing a specific concept in the "KNX Information Model" [03], such as, "urn:knx:dpa.352.51" (KNX Datapoint Annotation) or "urn:knx:fb.321" (*KNX Functional Block*). Multiple Resource Types may be included in the value of this parameter, each separated by a space. The maximum length of this parameter is 63 octets.

This clause explains the basic mechanisms, query formats, and other requirements that are common over all the discovery operations. More details per discovery type are given in later clauses. Note that all discovery types re-use the same generic mechanism of Link Format querying for attribute values, simplifying a constrained implementation.

2.6.1.3.2 Basic Query Format

A KNX IoT device SHALL support the query filtering even if this is designated as optional in [RFC7252] and [RFC6690]. The hostname is typically an IPv6 [RFC2460] literal between square brackets, either unicast address or multicast address. For example: [ff03::fd]. The below CoAP request format is used to do a one-attribute query in general:

```
GET coap://hostname/.well-known/core?attribute=value
```

2.6.1.3.3 Multiple Query-Attributes Format

Attributes MAY appear in any order; this order has no specific significance. Allowing multiple attributes extends [RFC6690], allowing only one attribute query. All the query attributes are logically AND-ed to provide the query result. Multiple query attributes MAY be applied for unicast and multicast queries, and the number of attributes for query is not limited. A KNX IoT device MAY impose limits on the number of query attributes to reduce complexity.

```
GET coap://hostname/.well-known/core?attribute=value&attr2=val2
```

2.6.1.3.4 Wildcard Usage and Hierarchy

Also, note that per [RFC6690], the "*"-character can be used as the wildcard in any query attribute value. *partialValue* is a partial value prefix string, which is used to search for multiple values that start with the string *partialValue* and end with the wildcard character "*".

```
GET coap://hostname/.well-known/core?attribute=partialValue*
```

2.6.1.3.5 Basic Response Format

A successful response to a query SHALL consist of a unicast 2.05 response with CoRE Link Format [RFC6690] payload containing the query result. The CoAP Token in the response SHALL be the same as the request Token. The exact attributes returned MAY depend on the query parameters used. If not explicitly specified, supporting concatenated query parameters is OPTIONAL. The </*path*> returned depends on the query type, i.e., if and what query parameters were given with the query request. The </*path*> returned is given by the device, i.e., it is device- and manufacturer specific. Note that secured, OSCORE, or coaps:// scheme links MAY be returned, although not shown in the following example.

In addition to the </*path*>, the response SHALL contain the Resource Type "rt" and Content-Type "ct" if the corresponding metadata are defined for the Point.

Discovery response
2.05 Content (Content-Format: application/link-format (40)) </*path*>; attr=value; attr2=value2, <other link 1>, <other link 2>

2.6.1.3.6 Query-Attributes

2.6.1.3.6.1 Endpoint Name "ep"

Endpoint Name provides a unique identifier within a defined domain. A domain might be a single installation of a site or the entire world (global). The value of the Endpoint Name attribute SHALL be a unique device identifier (e.g., *KNX Serial Number*). The response SHALL contain the requested ep identifier (*KNX Serial Number* and the *KNX Individual Address*) and the base path (e.g., <> or </api>}) for KNX IoT resources. A KNX IoT device SHALL ignore a multicast query request for "ep" if the queried Endpoint Name does not match its own or if the query request contains more than one "ep" attribute. A KNX IoT device SHALL accept and respond to a discovery request if its configured *KNX Individual Address* or built-in *KNX Serial Number* matches the *KNX Serial Number* or *KNX Individual Address* received with the discovery request (see also 2.4.2).

CoAP client discovers a device with a particular device identifier (e.g., *KNX Serial Number* from QR code scan) in a network.
REQ: GET coap://{ipv6-multicast}/.well-known/core?ep=knx://sn.1caffe1234 **RES:** 2.05 CONTENT (Content-Format: application/link-format (40)) **Payload:** <>;ep="knx://sn.1caffe1234 knx://ia.33a3.20a"

With the above-shown discovery request, a MaC will get the KNX IoT device's IP (unicast) address that is not part of the CoAP response payload but contained in the IP response frame. Using the returned IP address, the MaC may then do further (unicast) discovery or other unicast actions (e.g., device configuration)

2.6.1.3.6.2 Programming Mode "if.pm"

Suppose the *Programming Mode* is active (see clause 2.5.3.9), a KNX IoT device SHALL accept and respond to the discovery request to /.well-known/core and with query attribute if=urn:knx:if.pm, with its *KNX Serial Number* and the <*KNX Individual Address*> in the response's ep attribute (ep="knx://sn.<*KNX Serial Number*> knx://ia.<KNX Installation ID>.<*KNX Individual Address*>").

If the *Programming Mode* is used to identify the device, then the MaC sends a multicast request to the device, which returns the *KNX Serial Number* with its device IP address. Using the returned IP address, the MaC can set, for example, the *KNX Individual Address* of the device in a subsequent step.

Sleepy KNX IoT devices in *Programming Mode* SHALL reply to requests after a maximum of 2 seconds (default CoAP timeout) to have a reasonable reaction time on the MaC.

If the device is not in *Programming Mode*, then a KNX IoT device SHALL return with CoAP response code "4.04 Not Found" with an empty payload (unicast request) or ignore the multicast query request.

Discover devices that are in *Programming Mode.*
REQ: GET coap://[FF03::FD]/.well-known/core?if=urn:knx:if.pm **RES:** 2.05 CONTENT (Content-Format: application/link-format (40)) **Payload:** <>;ep="knx://sn.1caffe1234 knx://ia.33a3.20a"

1557

1558 2.6.1.3.6.3 Interface Description "if"

1559 An interface description describes a generic CoAP resource to interact with a Point or a set of Points. The
1560 interface provides a view of organized Points. The interface description "if" attribute is an opaque string
1561 indicating a specific interface definition defined in clause 2.5.3, "Interface Types (if)". The "if" column
1562 defines the interface description (if=) attribute value to be used in the CoRE Link Format for a resource
1563 conforming to that interface.

1564 The "urn:knx" namespace identifier SHALL be omitted in the response to reduce payload (see clause
1565 1.4.5, "Uniform Resource Name") since the request filter has already been applied to "urn:knx".

CoAP client discovers all (wildcard) output points on a specific device.
REQ: GET coap://{ipv6-unicast}/.well-known/core?if=urn:knx:if.o **RES:** 2.05 CONTENT (Content-Format: application/link-format (40)) **Payload:** </*point-path-example*>;rt=":dpa.352.55";if=":if.o";ct=50

1566

1567 The "if" attribute MAY be used with a query attribute to read a value from a device in a specified device
1568 state such as the *Programming Mode* (see clause 2.6.1.3.6.2).

1569 A MaC MAY use multicast or unicast requests to test the destination device with a specific *KNX Serial*
1570 *Number* if the KNX IoT device is currently in *Programming Mode*.

1571 A KNX IoT device SHALL support the concatenation of "ep=knx://sn.{*KNX Serial Number*}" and
1572 "if=urn:knx:if.pm" query parameters. The response SHALL contain the *KNX Serial Number* and the if all
1573 query parameters match.

CoAP client discovers (multicast) a device in *Programming Mode* (pm).
REQ: GET coap://[FF03::FD]/.well-known/core?ep=knx://sn.*&if=urn:knx:if.pm **RES:** 2.05 CONTENT (Content-Format: application/link-format (40)) **Payload:** <>;ep="knx://sn.1caffe1234 knx://ia.33a3.20a"

1574

The above example shows the combination of query attributes ep=knx://sn.*&if=urn:knx:if.pm in a multicast discovery request. The attribute if=urn:knx:if.pm here limits the number of responses to only those resources (devices) that have their *Programming Mode* enabled. Using query attribute ep=knx://sn.* only, i.e., wildcard on the serial number sn, would result in responses from all devices.

NOTE 6 Multiple devices could respond with their ep in the payload, i.e., if more than one device is present with their *Programming Mode* enabled.

2.6.1.3.6.4 Resource Type "rt"

A discovery client MAY use the resource type represented by the "rt" attribute (see [RFC6690] clause 3.1) for selected CoAP resources. Every discoverable resource SHALL have an "rt" attribute associated. The resource type value for a resource SHALL contain "urn:knx:" and is to be interpreted as an identifier for the type of the resource.

The "rt" attribute value is used to filter a discovery request based on a *Functional Block* type ID (e.g., urn:knx:fb.321) or a Datapoint Annotation (e.g., urn:knx:dpa.321.51). The "rt" member MAY contain a list of different types representing semantic information. The syntax of the value of this attribute is: rt={type}.

A Datapoint Annotation combines Object Type (*Functional Block* type ID) and Property Identifier, as described in KNX standard Volume 7. A Point MAY have one or many Datapoint Annotations; for example, an "if.i" (input interface) can have bindings to several "if.o" Points (output interface). Hence, the Datapoint Annotation is either the Point Object Type and Property ID (DP Address) or possible Point bindings (e.g., LTE InfoReport input). Datapoint Annotations for "if.i" and "if.o" are described in KNX standard Volume 7 in the "LTE-Mode" datapoint description (see Communication – DP Address).

In the following example, a client device wants to discover Points with a specific Datapoint Annotation on a server device. If the server device hosts such Points, then the server device SHALL reply with a list containing all resources, including access path, resource type, and content type. In addition, the response SHALL at least contain Points with the interface type "if.o" or "if.i" and "if.p" in case of an unassigned (peripheral) *Group Address* configuration Point.

The "urn:knx" namespace identifier SHALL be omitted in the response to reduce payload (see clause 1.4.5, "Uniform Resource Name") since the request filter has already been applied to "urn:knx".

CoAP client discovers all (wildcard) Heat Valve Actuator (Functional Block 352) points on a specific device.
REQ: GET coap://{ipv6-unicast}/.well-known/core?rt=urn:knx:dpa.352* **RES:** 2.05 CONTENT (Content-Format: application/link-format (40)) **Payload:** </*point-path-example1*>;rt=":dpa.352.51";if=":if.i";ct=50 60, </*point-path-example2*>;rt=":dpa.352.55"; if=":if.o";ct=50 60

2.6.1.3.6.5 Sector "d"

In a Resource Directory [RD] context, a sector is a logical grouping of resources. The abbreviation "d" is used for the sector in query parameters for compatibility with deployed implementations. The sector identifier "d" is used to discover resources reachable through a KNX *Group Address*. Therefore, the value of "d" represents a KNX *Group Address*. The syntax for a KNX *Group Address* is d=urn:knx:g.{group-type}.{group-address}. This specification defines "s" as {group-type} for S-Mode *Group Address* and "p" for unassigned (peripheral) tags as {group-type} (see KNX standard Volume 10/1 clause 5.3). However, other {group-types} MAY be added in the future.

The wildcard character ("*") is neither allowed in the <group-type> nor in the <group-address> field since this can result in large responses. The KNX IoT device SHALL return a response code of "4.00 Bad Request" if the request contains a wildcard character ("*") in a KNX *Group Address* discovery query.

The "urn:knx" namespace identifier SHALL be omitted in the response to reduce payload since the request filter has already been applied to "urn:knx".

CoAP client discovers points that belong to a specific *Group Address* (S-Mode) on a specific device.

REQ:
GET coap://{ipv6-unicast}/.well-known/core?d=urn:knx:g.s.1234

RES:
2.05 CONTENT (Content-Format: application/link-format (40))
Payload:
</*point-path-example*>;rt=":dpa.352.51";ct=60

A KNX IoT device SHALL support the concatenation of query parameters with resource type "rt", but only for "dpa" prefixed attributes (Datapoint Annotation), and the sector type "d" with {group-type} "p". The query attributes are logically AND-ed to provide the query result. This allows clients (e.g., controller *Functional Block*) to find corresponding runtime interworking inputs or outputs of a particular Datapoint Annotation. This discovery query is intended to be used in small ad hoc configurations, for example, in combination with a push-mode-like binding.

If "rt" and "d" are used in combination, then the wildcard character ("*") is neither allowed in "rt", nor in the "d" query parameter. The KNX IoT device SHALL return a response code of "4.00 Bad Request" if the request contains a wildcard character ("*").

CoAP client discovers (multicast) Points that belong to the peripheral *Group Address* = 1

REQ:
GET coap://[FF03::FD]/.well-known/core?rt=urn:knx:dpa.352.55&d=urn:knx:g.p.1

RES:
CONTENT (Content-Format: application/link-format (40))
Payload:
</*point-path-example*>;rt=":dpa.352.55"; if=":if.o";ct=50 60

2.6.1.4 Linkage and Resolution

Device resources MAY be identified by a Link Identifier and a relative path component. To identify a specific resource, a Logical Resource Identifier (LRI) is used; this LRI can be generated as a unique ID by the device. A Link Identifier is defined by the URI scheme: "knx".

The following LRI example shows a reference to a KNX IoT device resource of a particular KNX device. Leading zeros in the *KNX Serial Number*, KNX Installation ID, and *KNX Individual Address* SHALL be omitted.

- *KNX Serial Number* (ASCII hexadecimal lowercase letters): knx://sn.1caffe1234/{point-path}.
- *KNX Individual Address* (ASCII hexadecimal lowercase letters): knx://ia.33a3.20a/{point-path}.

The same ID SHALL be used as endpoint name (ep) to discover the device by clients in case the IP address has changed.

2.6.2 Device IP Address

2.6.2.1 Requirements

A single network MAY consist of a bridged Ethernet or Wi-Fi network, and all KNX IoT devices belong to the same IPv6 multicast domain. In this case, a link-local IPv6 addressing (LLA) is sufficient. A larger OT (Operational Technology) network usually contains stub networks. Stub networks, like a Thread network, require routable IPv6 addresses to communicate across networks, such as a unique-local identifier (ULA).

The Thread network uses an On-mesh prefix for link-local communication, which corresponds with an LLA prefix in a Wi-Fi or Ethernet network. A stub router (Border Router) connects the stub network with an adjacent network. The stub router SHALL provide its routable prefix if the network does not provide a routable prefix. A stub router SHOULD advertise routable prefixes on the adjacent network. A Thread Border Router, for example, SHOULD advertise routable prefixes on adjacent networks with Router Advertisements [RFC4861] with the Route Information Option [RFC4191].

The KNX IoT device SHALL listen on configured CoAP ports (see Device Resource Object clause 2.5.6). The default CoAP port configuration SHALL be used for unicast and multicast as defined in [RFC7252].

All KNX IoT devices SHALL have an IPv6 address. KNX IoT devices will use the unique-local identifier fd00::/8 according to [RFC4193] for communication within private networks (site or organization, but not in the global IPv6 Internet). Furthermore, a KNX IoT device SHALL support the configuration of at least three routable IPv6 addresses (e.g., KNX Fabric, KNX Installation, and Thread Domain prefix) in addition to the link-local address. As for any IPv6 networking device, multiple IPv6 addresses may be available. If available, the prefix having the unique-local identifier concatenated with a KNX Fabric ID (see configuration property "dev/fid" in clause 2.5.6) yields the system Unique Local Address (ULA) prefix that SHOULD be used to construct the default address for KNX IoT messages. For each KNX IoT Fabric that wants to use KNX-specific ULA for the deployed system, the MaC either gets a network-specific ULA configuration from the network administrator or generates a random, unique 40-bit KNX Fabric ID representing the Global ID according to [RFC4193]. This procedure is part of the configuration with a MaC and is not in the scope of this specification.

The interface identifier of a KNX IoT device SHALL be 64 bits long. It MAY be constructed by various methods [RFC4291, RFC7136, Thread Specification] but SHALL be persistent over reboots for the operational phase of the device. It MAY also be created randomly and recreated during a device reset to the default state. The device SHOULD implement some method of Duplicate Address Detection (DAD), e.g., [RFC4429].

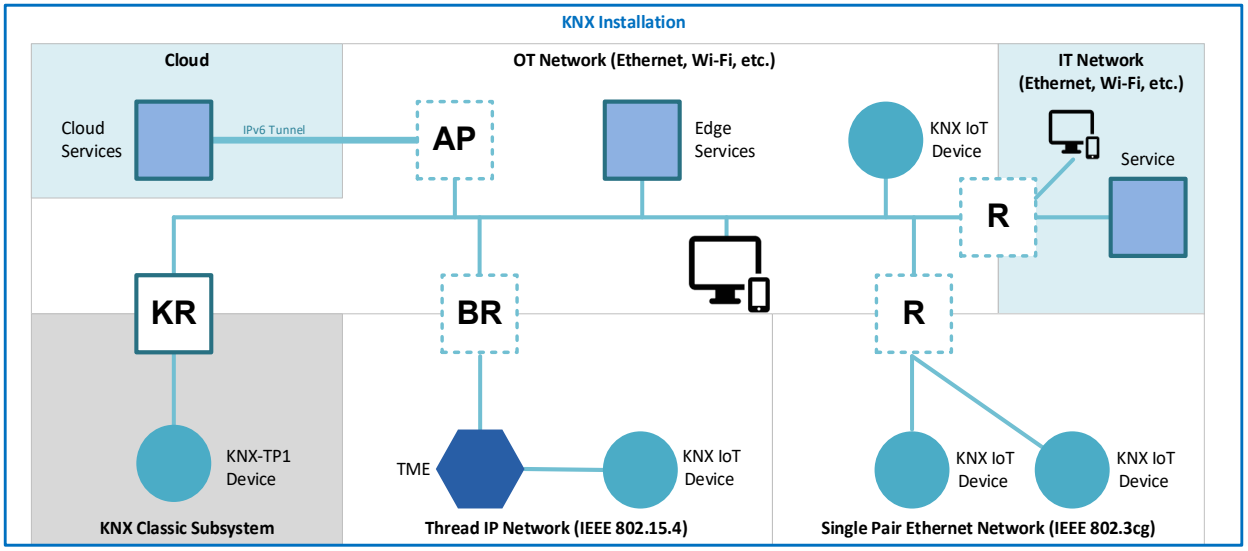
Every IPv6 subnet, such as a Thread or Wi-Fi network, gets a separate 16-bit subnet ID. Subnet ID "0000" is, by convention, reserved for administrative services. This results in the following IPv6 address format for a KNX IoT device when using the ULA:

ULA Routing Prefix /48		Subnet ID	Interface Identifier
Unique-Local	KNX Fabric ID / Global ID		
fd	11:2222:33a3:	0001:	6cd9:8ad2:8e88:1f68

2.6.2.2 KNX Network Example

The following KNX network example contains devices and services in different network segments. The KNX Fabric ID and KNX Installation ID simplify the routing table configuration instead of "whitelisting" each device individually. Hence the KNX Fabric ID and KNX Installation ID are part of the security zone concept (see clause 3.1), and access is only possible among devices and services with the same ULA. In addition, the KNX Fabric ID MAY be used to separate groups of devices within an OT network. For example, devices that belong to a tenant (apartment) or trade (lighting, fire protection, HVAC system, etc.) MAY have its IPv6 address range in larger buildings.

The example also contains an IPv6 tunnel to a Cloud service via Access Point (AP). This tunnel, or rather the service at the end of the tunnel, belongs to the KNX Fabric too. The same applies to services in an IT network (office network). In both cases, routers block requests to devices not belonging to the KNX Fabric.



AP: Access Point, BR: Border Router, KR: KNX IoT Router, R: IP Network Router, TME: Thread Mesh Extender

Figure 17 – KNX Network Example

2.6.3 Unicast Operation

2.6.3.1 IPv6 Unicast Port Number

The default port number for KNX IoT unicast communication SHALL be 5683 (CoAP default port number). It is RECOMMENDED not to change the default port for runtime communication (see 2.5.6, "Device Object Resource (dev)").

2.6.3.2 Options

Table 39 lists the Options for a server KNX IoT device MANDATORY ("M", for mandatory), SHOULD ("R", for recommended), or OPTIONAL ("O") support. The hyphen ("-") means not applicable, and the "X" means the Option can be used but SHOULD NOT be used in the indicated context. All the Options are specified in [RFC7252] except where noted otherwise in the "Notes" column. Any Option in a confirmable request that is not recognized by a server where the option number has the "Critical" flag [RFC7252] set SHALL lead to a "4.02 Bad Option" response code. These are the uneven Option numbers. Option support for a client is not specified in detail. However, it SHALL support at least the Options marked MANDATORY ("M") in the Server "In Response" column below.

Table 39 – Unicast operation server requirements

Opt Nr	Option Name	Server support for the Option		Notes
		In Request	In Response	
1	If-Match	O	-	
3	Uri-Host	O	-	Usually not included in a CoAP request; a Server SHALL respond 4.02 if it does not support multiple virtual servers per Clause 5.10.1 of [RFC7252].
4	ETag	O	O	

Opt Nr	Option Name	Server support for the Option		Notes
		In Request	In Response	
5	If-None-Match	O	-	
6	Observe	O	-	[RFC8323]
7	Uri-Port	O	-	Normally not included in a CoAP request; a Server SHALL respond 4.02 if it does not support multiple virtual servers per [RFC7252] clause 5.10.1.
8	Location-Path	-	M	Typically used when resources are created using POST.
9	OSCORE	M	M	OSCORE application layer security [RFC8613]
11	Uri-Path	M	-	
12	Content-Format	M	M	Applicable for PUT and POST in requests and GET in the response.
14	Max-Age	-	O	By default, excluded in response.
15	Uri-Query	M	-	
17	Accept	M	-	Applicable for PUT and POST if payload returned.
20	Location-Query	-	O	
23	Block2	O	O	[RFC7959]
27	Block1	O	O	[RFC7959]
28	Size2	O	O	[RFC7959]
35	Proxy-Uri	O	-	
39	Proxy-Scheme	O	-	
60	Size1	O	R O	"R" for Clause 5.9.2.9 of [RFC7252] defined Size1 semantics for 4.13 response; "O" for [RFC7959] Size1 semantics.
252	Echo	M	M	[RFC9175]
258	No-Response	R	-	[RFC7967]

2.6.3.3 Response Codes

2.6.3.3.1 Overview

Table 40 lists the response codes that the server KNX IoT device SHALL ("M", for MANDATORY) or MAY ("O", for optional) support. The "X" means SHOULD NOT use this response because there is a better alternative or no reason to use it. The usage rules for the code are given by [RFC7252] or the RFC listed in the "Notes" column, plus the "Requirements for Use" stated in this column, if any. The required support for a CoAP client is not specified in detail; however, a client SHALL at least be able to distinguish the basic CoAP response code classes 2.xx (success), 4.xx (client error), and 5.xx (server error).

Using error messages that do not provide implementation details is important to avoid information leakage. When a server encounters a problem, only standardized status codes of the 4.xx and 5.xx range SHALL be used consistently with their intended semantics.

A server MAY choose to stop processing as soon as a problem is encountered, or it MAY continue processing and encounter multiple problems. For instance, a server might process multiple attributes and then return multiple validation problems in a single response. When a server encounters multiple problems for a single request, the most generally applicable error code SHALL be used in the response.

In the case of multiple errors, the general priority of errors is manufacturer specific, and MAY differ amongst KNX IoT devices. However, response code "4.01 Unauthorized" has the highest priority and SHALL be used for clients that do not have valid authentication credentials (e.g. (D)TLS [X.509] certificate and/or OSCORE pre-shared key).

If no specific response code is applicable, "4.00 Bad Request" might be appropriate for 4.xx errors, or "5.00 Internal Server Error" might be appropriate for 5.xx errors.

Table 40 – Server response codes

Resp. Code	Description	Support Level	Notes and Requirements for Use
2.01	Created	**M**	"Create" operation, e.g., of an item, is completed successfully; [RFC7252].
2.02	Deleted	**M**	"Delete" operation is completed successfully; [RFC7252].
2.03	Valid	O	Only used when ETag Option was in request.
2.04	Changed	**M**	"Write" operation is completed successfully; [RFC7252].
2.05	Content	**M**	"Read", "Notification", or "Discover", etc. operation is completed successfully; [RFC7252].
2.31	Continue	O	This code indicates that the transfer of this block of the re quest body was successful and that the server encourage s sending further blocks; [RFC7959].
4.00	Bad Request	**M**	The response means the server could not understand the request due to invalid syntax; [RFC7252].
4.01	Unauthorized	**M**	Used if the client could not (yet) be properly authenticated to check if it is authorized to access this resource. The client SHOULD perform action(s) to authenticate itself before retrying the request. If a request results in several errors, then Unauthorized SHOULD have the highest priority.
4.02	Bad Option	**M**	Diagnostic payload SHALL be included and SHOULD start with the first unrecognized critical-option number as a 4-byte UTF-8 string.
4.03	Forbidden	**M**	Used if the client could be authenticated but does not have the authorization to access this resource (e.g., wrong access scope). The client SHOULD take action(s) to improve its authorization status before retrying this request.
4.04	Not Found	**M**	Server cannot find the requested resource; [RFC7252].
4.05	Method Not Allowed	**M**	The request method (GET or PUT etc.) is known by the server but has been disabled and cannot be used; [RFC7252].

Resp. Code	Description	Support Level	Notes and Requirements for Use
4.06	Not Acceptable	**M**	This response is sent when the server, after performing content negotiation, doesn't find any content following the criteria given by the client; [RFC7252].
4.08	Request Entity Incomplete	O	The code indicates one or more missing blocks. The server has not received the blocks of the request body that it needs to proceed; [RFC7959].
4.09	Conflict	X	See [RFC8132], typically not expected to be supported.
4.12	Precondition Failed	O	If 4.12 not implemented, then 4.02 SHALL be returned if a conditional request [RFC7252] is made.
4.13	Request Entity Too Large	O	"O" as defined in [RFC7252] for request size only.
4.15	Unsupported Content-Format	**M**	The media format of the requested data is not supported by the server, so the server is rejecting the request; [RFC7252].
4.22	Unprocessable Entity	X	See [RFC8132], typically not expected to be supported.
4.29	Too Many Requests	**M**	[RFC8516]
5.00	Internal Server Error	**M**	[RFC7252]
5.01	Not Implemented	X	[RFC7252] requires the use of 4.05 instead for all cases.
5.02	Bad Gateway	O	Only relevant for CoAP Proxy; [RFC7252].
5.03	Service Unavailable	O	The server is not ready to handle the request. Common causes are a server that is down for maintenance or overloaded. The server SHALL return a Retry-After header indicating how long (delay in seconds) the client should wait before making a follow-up request; [RFC7252].
5.04	Gateway Timeout	O	The request does not complete within a specified time. The request times out and returns a Gateway Timeout error. Only relevant for CoAP Proxy; [RFC7252].
5.05	Proxying Not Supported	O	If proxying is not supported for any resource, a CoAP request with Proxy-Uri Option SHOULD trigger a 4.02 response instead of 5.05.

2.6.3.3.2 Error Responses to Queries

This clause applies to all unicast operations in the context of Discovery, i.e., unicast interactions on the "well-known/core" resource. Error responses are generated when a unicast request is sent and an error occurs in processing the request. The applicable response codes, including error responses, are defined in [RFC7252] clause 12.1.2. Specifically, in the case where a query contains too many elements for the responding server to handle is relevant for Discovery. This includes, for example:

- Too many query parameters.
- Too long strings in query parameters or other parts of the URI.
- Too long URI in total.

In such situations, the server SHOULD respond with a response code "4.13 Request Entity Too Large". This might indicate to the client that it SHOULD adapt its query before sending it again. It might happen that the response to a discovery request (holds for all requests) would be too large to construct or send for the server. Without a specific error code for this situation, the server could indicate this with an error response code "4.00 Bad Request" to indicate that the client might change /restrict the discovery query.

Since discovery queries might lead to long answers, especially if filtering is not implemented fully, clients SHALL support the option of CoAP block-wise Transfers [RFC7959].

An exception is, however, when the CoAP server is only temporarily occupied and will soon (within some seconds) be available to process larger queries. Then, the server SHOULD respond with the response code "5.03 Service Unavailable".

2.6.3.3.3 Rate limiting

KNX IoT devices SHALL rate-limit requests from clients that are sending excessive requests. This prevents battery-powered devices, for example, from being discharged. Errors in software or configurations in other parts of the system unintentionally cause many load-based denial-of-service incidents in systems.

2.6.3.3.4 Block-wise Transfer

KNX IoT devices (server) that do not support CoAP block-wise transfers SHALL respond with response code "4.02 Bad Option" if a client requests a block-wise transfer in the request.

2.6.4 Multicast Operation

2.6.4.1 IPv6 Multicast Port Number

The default port number for KNX IoT multicast discovery SHALL be 5683 (CoAP default port number).

2.6.4.2 CoAP multicast scopes

IANA has reserved the IPv6 [RFC8200] address range ff0x::fd as the "All CoAP Nodes" variable scope multicast address [RFC7252]. The "x" designates the 4 bits that declare the multicast address scope; all scopes can be used in principle. This multicast address is used for CoAP service discovery; it SHALL NOT be used for other purposes. A device SHALL support CoAP multicast discovery by listening to this multicast address on the following scope:

Link-local (2):

- Typically used to query in a single Wi-Fi or Ethernet network segment.

Realm-local (3):

- Typically used to query in a single mesh topology IPv6 network.

NOTE 7 Realm-local is defined for 6LoWPAN and is currently only implemented in certain IPv6 mesh network standards, e.g., Thread.

- A LAN or Wi-Fi node SHOULD NOT send realm-local multicast messages.

Realm-local (3) SHALL be the default scope to discover all nodes in a meshed network, i.e., across routing (on the same physical media) devices.

Site-local (5):

- Site-local is the default in an installation and spans across networks and stubs (Wi-Fi, Ethernet, and, Thread network segments).

2.6.4.3 Response suppression

If a query component is not present in a multicast request to "/.well-known/core", the receiving CoAP server SHALL NOT respond to the query in compliance with [RFC6690] and [RFC7252]. If there is no matching result, e.g., the query result set has zero elements, and the request was sent as multicast, then a response to the query request SHALL NOT be sent. This behavior complies with [RFC6690] clause 4.1. If a multicast request was sent, any error responses (CoAP 4.xx or 5.xx class) SHALL be suppressed by the responding CoAP server. Only in the unicast case are error responses generated.

2.6.4.4 Response timing

The CoAP server SHOULD delay any response to a multicast query by a randomly chosen duration between 0 and DEFAULT_LEISURE (5 seconds, [RFC7252]).

2.6.4.5 Sleepy devices

Infrastructure devices (e.g., Thread Routing devices) have limited memory to cache messages on behalf of sleepy devices which have turned off the radio when idle to preserve battery lifetime. Reliable multicast communication to "sleepy" devices is not possible. Therefore, the communication patterns between devices SHOULD be designed so that sleepy devices are always the initiator of multicast messages.

Sleepy KNX IoT devices that have received a request SHALL reply to requests within 2 seconds for at least one wake-up cycle. It is RECOMMENDED to increase linear wake-up cycles to have a reasonable reaction time on subsequent requests from clients.

2.6.5 Multicast Group IP Addresses

IPv6 Multicast *Group Address* is constructed using unicast-prefix-based multicast addresses [RFC3306, RFC3307] comprising:

- The multicast prefix FF3x [RFC3306, Chapter 4], with FF3 indicating a multicast address assigned based on the ULA routing prefix and 00PT (0b0011 = 0x3) encoding the type of the following ULA routing prefix. x is any valid scope identifier defined in [RFC4291 and IANA registry], but SHALL NOT exceed the scope of the embedded ULA routing prefix [RFC3306].
- The ULA routing prefix (0xFD + 40-bit KNX Installation ID),
- and a 32-bit group identifier in the range 0x80000000 to 0xFFFFFFFF [RFC3307, Chapter 4.3]

The KNX Installation ID (see configuration property "dev/iid" in clause 2.5.6) SHALL be used for the system Unique Local Address (ULA) routing prefix.

In a Thread network, for example, this multicast address type enables IP header compression down to 48 bits [RFC6282] if the constrained stub network uses less than 16 compressible prefixes in total [e.g., Thread Specification, 6LoWPAN Contexts]. Therefore, for multicast group notifications among KNX IoT devices, the compressed address SHALL take the form of ff3x::00:<group id> [RFC6282, Chapter 3.1.1], with the ULA routing prefix and its type PT mapped to a context managed by the constrained stub network.

Note that the lifetime of a unicast-prefix-based multicast address SHOULD NOT exceed the Valid Lifetime field in the Prefix Information option, corresponding to the ULA routing prefix being used, contained in the Neighbor Discovery Router Advertisement message [RFC3306].

The group identifier SHALL be allocated either by a server (e.g., MaC) or the host (KNX IoT device), following the rules for server or host allocation referenced in [RFC3307]. The Function Point Table contains the configuration for the group identifier (see clause 2.5.7[OBJ]).

For example, the multicast groups SHALL be registered at the Thread Border Router to protect constrained KNX IoT devices from undesired network loads. The multicast groups MAY be registered through a configuration step, e.g., by the MaC, or the KNX IoT device MAY subscribe and maintain the groups of interest by itself [Thread 1.2 Specification, Chapter 5.24]. In a Thread network, for example, Rx-off-when-idle KNX IoT devices SHALL register multicast addresses with their Thread Parent to receive them via link-layer unicast transmissions. Multicast packets of scope admin-local (x=4), scope site-local (x=5), and higher will be forwarded by a backbone router function of a Border Router [Thread 1.2 Specification Chapter 9.4.7] in its default configuration if the multicast group is registered in the border routers multicast listener table.

NOTE 8 Source-Specific Multicast (SSM) is not supported for Thread Devices.

Example site-local scoped multicast address: FF35::30:<ULA-routing-prefix>::<group id or ga>

This results in the following IPv6 address format (example):

Multicast prefix	ULA routing prefix	Group Identifier
FF35:0030:	FD11:2222:33a3::	8000:0068

2.6.6 Message Flow Control

Flow control is used to limit messages between KNX IoT devices to prevent devices from being overwhelmed with data. Publisher flow control is a mechanism that pushes the blocking behavior to the data producer. When the message queue is full, the receiving device notifies the Publisher device to stop sending new messages for a defined time.

Suppose a KNX IoT device cannot serve a certain Publisher that is sending messages too fast; the device SHOULD respond with a unicast message response code "4.29 Too Many Requests" and set the Max-Age option to indicate the number of milliseconds after which the Publisher can retry.

The time to wait in Max-Age SHOULD be enough to empty the message queue on the receiving KNX IoT device.

If a Publisher device receives the "4.29 Too Many Requests" response code from a KNX IoT device for a publish message, it SHALL NOT send new publish messages before the time indicated in Max-Age has passed.

2.6.7 Creating, Updating, and Deleting Resources

2.6.7.1 Introduction

A MaC can create new resources and modify or delete existing resources (see clause 2.4.3.3 "Partial/Differential download"). However, a KNX IoT device MAY NOT check configurations before applying (in a single or multiple "transaction"). Hence, the MaC SHALL write only correct and consistent data to a KNX IoT device.

2.6.7.2 Creating resources

For example, a Function Point Table or Access Control List resource can be created by sending a POST request to a resource representing a collection of resources (e.g., fp/g, fp/r, or fp/p). The request contains an array of resource objects as primary data. The resource object SHALL contain an ID member created by the MaC. An ID SHALL be specified with an "id" key, the ID SHALL be unique on the KNX IoT device.

If a POST request has been created successfully, the server SHALL return a "2.01 Created" response code.

Write a list of items into the Function Point Table.

REQ:
POST coap://{ipv6-unicast}/fp/g
(Content-Format: application/cbor (60)), OSCORE(code(POST), kid(<osc-id>)))
Payload:

```
[
  {
    0: 13,                //id
    11: "/ldsb1/soo",     //href
    7: [2305, 2401],      //ga
    8: 216                //cflag: 0b11011000
  },
  {
    0: 14,                //id
    11: "/ldsb1/rsc",     //href
    7: [2306],            //ga
    8: 64                 //cflag: 0b01000000
  }
]
```

2.6.7.3 Updating resources

If a request does not include all the members for a resource, the server SHALL interpret the missing attributes as if they were included with their current values. The server SHALL NOT interpret missing attributes as null values. For example, the following POST request is interpreted as a request to update only the "href" member in the Group Object Table:

Write Function Point Table item.

REQ:
POST coap://{ipv6-unicast}/fp/g
(Content-Format: application/cbor (60)), OSCORE(code(POST), kid(<osc-id>)))
Payload:

```
[
  {
    0: 1,                 //id
    11:"/ldsb1/soo"       //href
  }
]
```

If an update is successful and the server does not update any attributes besides those provided, the server SHALL return a "2.04 Changed" response code with no response document. When processing a request to modify a resource that does not exist, a server SHALL return response code "4.04 Not Found" with an empty payload.

2.6.7.4 Deleting Resources

An individual resource can be deleted by making a DELETE request to the resource's URL:

Delete Function Point Table item.

REQ:
POST coap://{ipv6-unicast}/fp/g/1
(OSCORE(code(DELETE), kid(<osc-id>)))

A server SHALL return a "2.02 Deleted" response code if a deletion request is successful with an empty payload.

A server SHALL return a "4.04 Not Found" response code if a deletion request fails due to the resource not existing.

It is also possible to delete a collection of items from a resource list with a single POST request, for example, on fp/g, fp/r, or fp/p, by sending just the corresponding "id" without additional attributes.

Delete a list of Group Object Table items.

REQ:
POST coap://{ipv6-unicast}/fp/g
(Content-Format: application/cbor (60)), OSCORE(code(POST), kid(<osc-id>)))
Payload:

```
[
  {
    0: 13        //id
  },
  {
    0: 14        //id
  }
]
```

A server SHALL return a "2.04 Changed" response code with no response document in the case all requested items are not present on the device anymore.

2.6.8 Pagination

2.6.8.1 Basic principle

A KNX IoT device (server) MAY want to limit the number of items returned in response to a subset ("page") of the whole set available. The page query parameter family "p" (e.g., "pn" or "ps") is reserved for pagination, and servers and clients SHALL use this key for pagination operations. A KNX IoT device SHALL support Linked List Pagination on all linked list resources (application/link-format).

With the query parameter "pn={value}" the client can request the number of pages to skip before starting to collect the result set. The page size "ps" defines the number of items per page and is a fixed configuration on a KNX IoT device (server).

If the request contains "ps", but the query parameter is not supported on the resource, then the server SHALL return a response code of "4.00 Bad Request".

Single items or pages can be accessed with pagination query parameters starting with "pn=0" if no query parameter is present.

Any link that could return a large payload (e.g., larger than min UDP frame size), potentially unbounded list of items in its GET response, SHALL implement the "pn" query parameter for pagination using the pattern described in the following example:

Pagination example that starts with the first list item (list size = 2 items).

REQ:
GET coap://{ipv6-unicast}/f/rts?pn=0

RES:
2.05 CONTENT (Content-Format: application/link-format (40))
Payload:
</p/rts/temproom>;rt=":dpa.321.51";ct=60,
</p/rts/tempcorrvalue>;rt=":dpa.321.111";ct=60

2.6.8.2 List Metadata Query Parameter "l"

List metadata query parameters are used to provide additional information about the list, such as the list size or default page size. The "l" query parameter SHALL be used exclusively, and a combination with other query parameters on the same resource is not possible.

To retrieve the value of {list-metadata}, a KNX IoT device SHALL accept a GET request to the URI {list-path}?l={list-metadata}&l={list-metadata}. If at least one of the requested {list-metadata} is supported, then the device SHALL return a response with CoAP response code "2.05 Content". The response payload SHALL contain the {list-metadata} as key and the {metadata-value} of the specified metadata as value. The content format is "application/link-format". The {point-path} in the response SHALL NOT contain query parameters.

If none of the requested {list-metadata} is supported, the device SHALL return a response with CoAP response code "4.04 Not Found" with an empty payload.

Metadata read example for a selected list metadata member (list size).

REQ:
GET coap://{ipv6-unicast}/fp/r?l=total&l=ps

RES:
2.05 CONTENT (Content-Format: application/link-format (40))
Payload:
</fp/r>;total=22;ps=5

KNX IoT defines a standard basic set of list metadata that MAY be applied to any Point. Standardized KNX IoT list metadata attributes names (key) do not contain a namespace. The following table defines the list metadata attributes for KNX IoT that the server MANDATORY ("M") or OPTIONAL ("O") supports:

Table 41 – List metadata member

Metadata Member	Resource Name	Method	Support	JSON Type	Notes
Total	total	GET	**M**	Number	The total number of existing items in the whole list.

Metadata Member	Resource Name	Method	Support	JSON Type	Notes
Page Size	ps	GET	**M**	Number	The default page size for pagination

2.6.8.3 Link List

A client MAY have to read multiple pages to get the full list of an application/link-format resource ("if.ll").

Suppose the list has more than one page than "rt=p.next" SHALL indicate that more requests (pages) are needed to get the full list. The link in the <brackets> contains the next page the client can request.

The client in the following example reads a resource list without the "p" pagination query parameter in the first request. The client knows that there is more to read because of the "rt=p.next" item. Hence, the client requests the next page with "pn=1".

Link list pagination example.

REQ:
GET coap://{ipv6-unicast}/p

RES:
2.05 CONTENT (Content-Format: application/link-format (40))
Payload:
</p/1>;rt=":dpa.x.y";ct=60,
</p/2>;rt=":dpa.x.z";ct=60,
</p/3>;rt=":mdt.siemens.x.y";ct=60,
</p?pn=1>;rt="p.next";ct=40

REQ:
GET coap://{ipv6-unicast}/p?pn=1

RES:
2.05 CONTENT (Content-Format: application/link-format (40))
Payload:
</p/4> rt=":dpa.x.y";ct=60,
</p/5> rt=":dpa.x.y";ct=60

2.6.9 S-Mode Group Communication

2.6.9.1 Publish Group Notification – Unacknowledged Multicast

In a brokerless system, a KNX IoT device MAY publish group notifications via IP multicast communication. The predefined default path structure ({messaging-path}) for S-Mode group notifications on IP multicast SHALL be ".knx".

A KNX IoT device MAY support the configuration of the path structure. The MaC (Management Client) configures the path structure for an installation. In some cases, the configured path structure may differ from the predefined structure, for example, "kitchen/sensors/" (see clause 2.5.7.4.2 "Function Point Recipient Resource Object").

CoAP [RFC7252] message fields SHALL be protected by OSCORE [RFC8613]. According to [RFC8613] (clause 4.2.), the code (PUT/GET/POST, etc.) is encrypted and integrity protected to prevent an intermediary from eavesdropping or manipulating the code (e.g., changing from GET to DELETE).

KNX IoT multicast messages SHALL be encrypted and integrity protected. Request and response codes are encrypted by OSCORE; see clause 3.6, "OSCORE Application Layer Security". Only dummy codes (POST/Changed) are visible in the header of the OSCORE message.

KNX IoT group notifications SHALL use POST inside the protected OSCORE message.

In contrast to normal Point read/write communication (unicast), the MaC MAY configure the same OSCORE credentials on multiple KNX IoT devices for S-Mode group communication. Therefore, an S-Mode OSCORE multicast message SHALL contain the "kid_context" to avoid duplicating message content. In addition, the KNX IoT message SHALL contain a "kid" with the corresponding OSCORE input material identifier (osc-id) configuration in the access token.

The MaC SHALL configure for each publisher KNX IoT device a unique "contextId" in the access token. The "contextId" SHALL be unique among access tokens with the same OSCORE input material identifier (osc-id). The Publisher KNX IoT device SHALL use the "contextId" for the "kid_context" in the OSCORE message, and the Recipient KNX IoT device SHALL check the "kid" with the related access token in the access control list.

PUBLISH Multicast S-Mode Group Message.
REQ: POST coap://{ipv6-multicast}/{messaging-path} (Content-Format: application/cbor (60), OSCORE(code(POST), kid(<osc-id>), kid_context(<contextID>))) **Payload:** { 4: <*KNX Individual Address*>, 5: { 6: <st>, 7: <ga>, 1: <value> } }

2.6.9.2 Publish Group Notification – Acknowledged Unicast

For reliable unicast message transfer, the Publisher KNX IoT device SHALL use a confirmed transport channel to a message broker or a Subscriber KNX IoT device, secured by OSCORE or, if supported, by (D)TLS (coaps://).

The destination resource ({messaging-path}) SHALL be of type "rt=urn:knx:g.s". A Publisher KNX IoT device publishing to a Recipient device MAY indicate the maximum lifetime of the value by including the Max-Age option in the publish request. The Recipient KNX IoT device SHALL return a response code of "2.04 Changed" if the publish request is accepted.

The Recipient KNX IoT device SHALL accept CoAP requests using the POST method.

Unicast S-Mode Group Message with (D)TLS destination URI from the Function Point Table.
REQ: POST coaps://{messaging-destination-uri}/{messaging-path} **Payload:** { 4: <*KNX Individual Address*>, 5: { 6: <st>, 7: <ga>, 1: <value> } }

For OSCORE application layer security, the KNX IoT device SHALL use credentials from the corresponding OSCORE access token, configured by the MaC and by thus known by all involved peers.

The MaC MAY configure a direct unicast message exchange in the Function Point Recipient Table. The resulting KNX IoT unicast messages SHALL contain a "kid" with the OSCORE input material identifier (osc-id) configuration in the access token.

It is RECOMMENDED to configure new OSCORE credentials for each unicast S-Mode group communication between two KNX IoT devices. The S-Mode OSCORE unicast message SHALL contain the "kid_context" to avoid duplicating message content (see clause 3.6.4).

After a device reset to a default configuration state, the MaC MAY uses the same OSCORE master secret ("ms") for device re-configuration. However, the MaC SHALL NOT reuse the "kid_context" and configure a new "kid_context". The client KNX IoT device SHALL use the new configured "contextId" for the "kid_context" in the OSCORE message in combination with the OSCORE message sequence number starting again at 0.

Unicast S-Mode Group Message with OSCORE destination URI from the Function Point Table.
REQ: POST coap://{messaging-destination-uri}/{messaging-path} (Content-Format: application/cbor (60), OSCORE(code(POST), kid(<osc-id>), kid_context(<contextID>))) **Payload:** { 4: <*KNX Individual Address*>, 5: { 6: <st>, 7: <ga>, 1: <value> } }

2.6.9.3 Group Notification Subscription – Acknowledged Unicast

A KNX IoT device SHALL allow OSCORE clients, and if supported (D)TLS clients, to subscribe to a resource of type "rt=urn:knx:g.s" using the CoAP observe mechanism. The KNX IoT device adds the Subscriber device to a list of Subscribers. After a successful subscription, the KNX IoT device sends Group Message events to all devices in the list of Subscribers (e.g., monitoring device, Message Broker, or KNX IoT Router).

The MaC MAY configure a subscription in the Function Point Publisher Table. The resulting KNX IoT unicast messages SHALL contain a "kid" with the OSCORE input material identifier (osc-id) configuration in the access token.

It is RECOMMENDED to configure new OSCORE credentials for each unicast S-Mode group communication between two KNX IoT devices (see clause 2.6.9.2).

Group messages from an intermediate service (e.g., KNX IoT Router) or a KNX IoT device SHALL contain the *KNX Individual Address* (sia) from the origin KNX device in the payload.

PUBLISH S-Mode Group Message (destination URI from the Function Point Table).
REQ: POST coap://{messaging-destination-uri} (Content-Format: application/json (50), OSCORE(code(POST), kid(<osc-id>), kid_context(<contextID>)) **Payload:** { "sia": <*KNX Individual Address*>, "s": { "st": <st>, "ga": <ga>, "value": <value> } } **RES:** 2.04 CHANGED

2.6.9.4 IP Address Resolving

The MaC usually configures the Function Point Table with device representations for runtime communication, such as a *KNX Individual Address* or an FQDN. Before a KNX IoT device can send a message, the device SHALL resolve the device representation to an IP address. If the KNX IoT device can successfully resolve an IP address, then the device SHALL store the IP address locally in non-volatile memory. The locally stored IP address SHOULD only be replaced once the re-resolution is successful. The following IP address resolving steps are necessary:

Table 42 – IP address resolving steps

Step	Procedure	Description
1	MaC configures Funcation Point Table.	The MaC configures a destination *KNX Individual Address*, FQDN, or IP address for S-Mode runtime communication.
2	Initial IP address resolving.	If no IP address has been configured and the Device Load State Machine is "loaded", then the device SHOULD try to discover the IP address (see "Discover IP address" procedure). This step SHALL be delayed (jitter 0..60 sec.) to avoid network congestion, for example, after the device restarts or Device Load State Machine changes to "loaded".
3	S-Mode communication.	If no IP address is available, the device SHALL discover the IP address (see "Discover IP address" procedure). Suppose the device already has an IP address, but the destination device is unreachable, or a "server not found error" occurs. In that case, the device SHOULD continue with the "Discover IP address" procedure.
4	Discover IP address	The KNX IoT device discovers a commissioned KNX IoT device with at least security and a *KNX Individual Address* configured (see 2.6.1, Discovery). After a successful IP resolution step, the KNX IoT device changes the IP address in non-volatile memory.

2.6.10 Point Publish/Subscribe

2.6.10.1 Subscriptions

CoAP observe is the default subscription mechanism for KNX IoT. Hence, resources that support subscriptions SHALL support CoAP observe [RFC8323].

Points that implement a resource of type "rt=urn:knx:g.s" (/.knx) SHALL support CoAP observe. The definition of other Points that support CoAP observe is manufacturer specific.

A client SHALL use a CoAP observe (GET) in combination with the "lt" query parameter for device Point subscriptions.

The KNX IoT device SHALL stop notifications after the configured subscription lifetime "lt".

When a client wants to subscribe to resources, for example, with a non-confirmable request on KNX IoT sleepy devices, the CoAP Token is important for the binding between requests and responses. CoAP [RFC7252] clause 3.5.1 does not give stricter guidelines than that the CoAP Tokens currently "in use" SHOULD (not SHALL) be unique. However, a KNX IoT client and server SHALL make sure that CoAP Tokens are not used in a way so that responses (or notifications) risk being associated with the wrong request.

Therefore, a client SHALL use new CoAP Tokens to re-subscribe on KNX IoT devices. The KNX IoT device (server) SHALL replace existing subscriptions based on the source IP address and IP port but SHALL NOT use the CoAP Token as an identifier for re-subscriptions.

2.6.10.2 Notifications

A notification event SHALL be sent to the subscriber IP address when the heartbeat time has elapsed since the previous message or the configured change of value (CoV) has exceeded the delta of the same value (see also clause 2.5.11.2, "Point Value Update Notification").

The notification heartbeat is indicated with the Max-Age option in the notification message. Suppose no update is received within this heartbeat period. In this case, the Subscriber SHOULD invalidate the last received value, and the Subscriber SHOULD behave according to manufacturer or application-specific requirements that are not part of this specification.

3 Security

3.1 Introduction

The following clauses describe a common approach in enabling a unified, multi-vendor building-automation solution meeting the technical requirements for a [62443-3-3] security Level 3 compliant installation. However, it remains the network administrator's responsibility to decide which security level an installation needs, and which infrastructure is needed to reach the security level (e.g., LDevID, a local Registrar/Domain CA, or a local KDC, etc.).

An installation involves many parties to successfully operate a building automation system. Each party has a specific role in managing an installation. There are the following roles relevant to security management in an installation:

- manufacturers of KNX IoT devices,
- network administrators who are responsible for the secure operation of the overall network infrastructure,
- system integrators who customize KNX IoT devices, integrate them into an installation, and perform commissioning,
- facility managers who monitor the system during their normal operation and respond to alarms,
- and service technicians who are responsible for maintaining and repairing the installation.

Successful function of an installation is only possible when KNX IoT devices are correctly commissioned, operated, and maintained. Access scopes have been defined with these roles in mind, however, the mapping of roles and permissions to access scopes are out-of-scope of this document. Another key aspect of this security specification is the need to strengthen the trust relationship between the manufacturer and the operator by trusting the manufacturer device certificate IDevID (Initial Device ID).

The solution can be deployed on top of any IP network deployment, providing uniform communication infrastructure independent of underlying networking technologies (e.g., Ethernet, Wi-Fi, or a Thread-based IEEE 802.15.4 network). At the transport level, a security zone concept based on IPv6 network segments (ULA prefix) and X.509 certificates can be configured that reflect diversely administered systems. A KNX IoT device may be a member of multiple security zones. On top of that, the application-level authorization of the Resource model limits the access scope of what a KNX IoT device is entitled to do within the scope of a single security zone. The following clauses describe a PAKE (Password Authenticated Key Exchange) and X.509 variants of operational device certificate (LDevID) enrollment procedures:

- Simple Enrollment (Pull Certificate): A user with an on-premises MaC configures Registrar settings on a KNX IoT device. Subsequently, the KNX IoT device automatically enrolls the LDevID with a Registrar with the EST-CoAPS [RFC9148] protocol. The resulting LDevID has a finite lifetime.
- Enrollment with MaC (Push Certificate): A user with an on-premises MaC configures the KNX IoT device LDevID. This enrollment procedure is usually used if the network infrastructure does not provide a Registrar service. Hence, the LDevID most probably has an infinite lifetime.
- Authentication code-based enrollment: A user provides out-of-band a setup key (e.g., QR code, NFC, etc.) that all involved peers know. The resulting secure session between peers is used for authentication and access token configuration.

3.2 Device Identity Enrollment

3.2.1 Common Requirements

When the new device is turned on, it looks for open networks for device enrollment. Once a network is discovered, the device is ready for enrollment. A KNX IoT device MAY implement the simple enrollment concept as defined in "EST over secure CoAP" [RFC9148].

Before a device can join an installation or network (security zone), it SHALL be authenticated. However, not all KNX IoT devices can perform EST enrollment and have an IDevID. Therefore, a KNX IoT device SHALL support at least the PAKE method (see clause 3.6.2).

The following clauses describe the general device identity enrollment flow of a new device that wants to join a network.

A KNX IoT device MAY be provisioned with a domain CA Trust Anchor and an operational device certificate (LDevID). The process involves the following steps:

- Device Authentication,
- Domain CA Trust Anchor Provisioning,
- a MaC (maybe) triggers the device to start the Operational Device Certificate (LDevID) Enrollment.

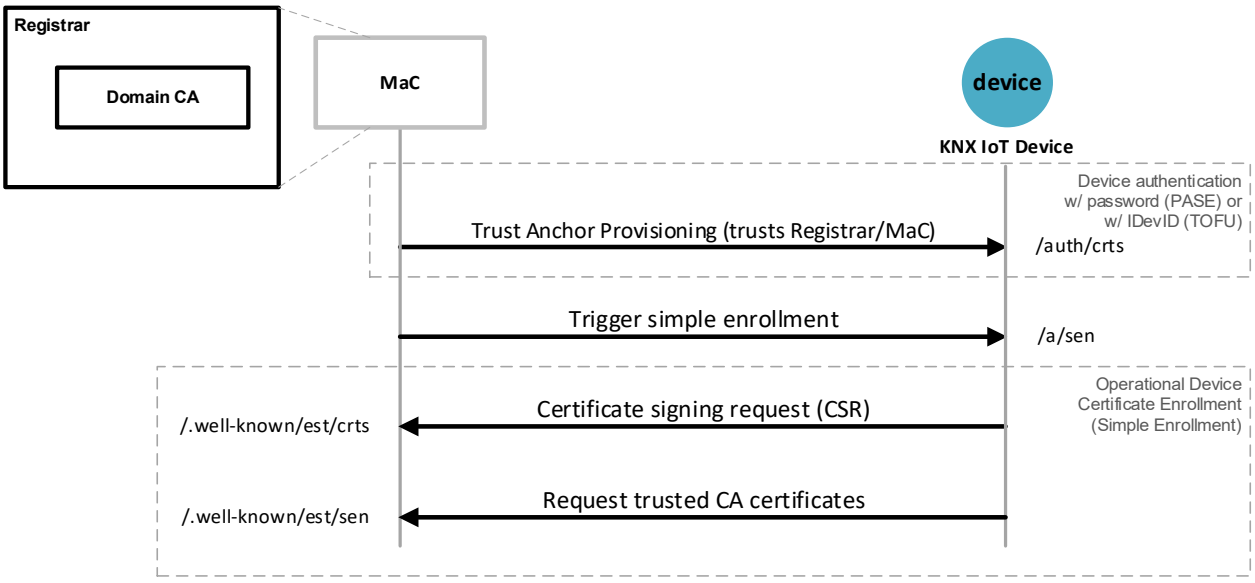


Figure 18 – Device Identity Enrollment

3.2.2 Device Authentication

In the first step, a secure channel between a KNX IoT device and the Registrar SHALL be established. This can be initiated by a MaC/Registrar (PUSH with PAKE) or by the device with simple enrollment (PULL).

For the simple enrollment, the KNX IoT device without LDevID SHALL use the IDevID as a client certificate for the mutual authenticated (D)TLS session with the Registrar. IDevID is a factory-imprinted certificate and serves as an initial proof of identity. If the Registrar is not accessible in the local network, the device MAY use a Proxy acting as a relay. The proxy function will vary based on L2 technology. For Thread-based devices, it might be (for instance) a border router, and for wired Ethernet, it might be either null or the switch. Hence, KNX IoT devices that support EST enrollment SHALL support brski-proxy and brski-registrar DNS-SD service names as defined in [RFC8995] clause 8.6 for Registrar discovery. The (D)TLS handshake SHALL perform mutual authentication based on the device manufacturer and Registrar certificates. The device lacks yet the domain CA to verify the Registrar certificate. Therefore, it SHALL provisionally accept the Registrar certificate to complete the (D)TLS handshake. At this stage, the Registrar MAY decide not to continue with the process if the KNX IoT device is not authorized to join the security zone (e.g., verification of the device Manufacturer certificate failed).

3.2.3 Domain CA Provisioning

A typical KNX IoT device contains no pre-configured operational trusts (CA certificates in the trust list) before first commissioning. In the second step, the Registrar configures the domain CA certificate on the device. In KNX IoT, the Domain CA is an operational Trust Anchor that:

- issues operational device certificates (LDevID) for a particular KNX IoT device and certifies the ownership of a public key by the named subject of the KNX IoT device certificate to a security zone or, rather, an installation domain.
- might be a local service entity such as a part of a MaC or is a remote service entity from a manufacturer or enterprise network.

3.2.4 Operational Device Certificate Enrollment (Pull Certificate)

3.2.4.1 Requirements

The operational device certificate enrollment (PULL) requires that the new KNX IoT device first discovers the Registrar either by using DNS, mDNS, or the KNX IoT device has a configuration setting where to find the Registrar.

In the next step, the KNX IoT device sends a Certificate Signing Request (CSR) to the Registrar. [RFC5967] defines the MANDATORY format for a CSR.

A CSR MAY include all of the key details of the requested certificate, such as subject, organization, and state. The CSR SHALL include the specific subject distinguished name from the manufacturer device certificate (IDevID) and the LDevID certificate's public key to get signed. The private key SHALL NOT leave the device at any time and SHALL NOT be readable via MaC or any other (remote) communication connection.

After sending the CSR, the Registrar MAY decide not to continue the process if the new KNX IoT device cannot be authorized automatically. The Registrar receiving this CSR SHOULD validate the proof-of-identity and then check the subject distinguished name included in the CSR with the manufacturer device certificate (IDevID). The CSR only gets signed by the Trust Anchor if all checks are successful.

A successful response SHALL be a certs-only [CMC] Simple PKI Response (PKCS#7), as defined in [RFC5273]. The authorization at the Registrar is deployment specific and may need manual confirmation from a network administrator. As soon as the new KNX IoT device is authorized to join the domain, the Registrar returns the LDevID to the KNX IoT device. The returned LDevID SHALL be a security zone (e.g., installation domain) specific [X.509] device certificate.

The enrollment process for a new LDevID is based on EST-CoAPS (Enrollment over Secure Transport) [RFC9148]. Devices MAY re-enroll via EST at any time before certificate expiration but SHALL re-enroll with their IDevID after certificate expiration.

3.2.4.2 LDevID Simple Enrollment Command (a/sen)

A MaC MAY start the LDevID enrollment; the KNX IoT device SHALL return a response with CoAP response code "2.04 Changed" if the command was accepted.

Table 43 defines the resource for triggering the LDevID enrollment with EST-CoAPS and a device MANDATORY ("M") support.

Table 43 – LDevID enrollment resource

Resource path	Resource Types (rt)	Format	Method	Support	Request/Response	Notes
a/sen	NA	cbor	POST	**M**	Req: Content-Format: application/cbor Payload: { 2: "renew" } Res: 2.04 CHANGED	Starts the X.509 certificate enrollment.

3.2.4.3 LDevID Simple Enrollment Resource Object

Table 44 specifies the MANDATORY ("M") attributes and defines JSON keys and the CBOR mapping for CoAP the request object to a/sen:

Table 44 – LDevID Simple Enrollment Resource Object members

JSON Key	CBOR Key	CBOR Type	Support	Name
"cmd"	2	text string	**M**	Command for triggering or stopping the LDevID simple enrollment. Enum: renew, stop

3.2.5 Management Client as Registrar (Push Certificate)

When a Registrar is unavailable in the installation, a Management Client (MaC) MAY serve this function. Once a Registrar becomes available in the installation, devices will enroll with the IDevID described above. For transition and mobility purposes, KNX IoT devices may continue to use any actively enrolled certificate for communication until that certificate expires or is removed. Devices already using an LDevID SHALL NOT make use of new LDevIDs issued from different Registrars until configured to do so to avoid confusion in certificate selection.

In the EST-CoAPS certificate enrollment flow, the KNX IoT device is the client and sends requests to the Registrar (PULL). In a PUSH enrollment, the roles are changed. The MaC always acts as a client, controls the communication flow with KNX IoT devices, and sends requests to the KNX IoT device. The LDevID configuration is an additional step in the access token enrollment (see clause 3.6, "OSCORE Application Layer Security"). A KNX IoT device in default configuration state SHALL accept client (D)TLS sessions provisionally until the MaC has configured a domain CA.

The KNX IoT devIce MAY begin the generation of a key pair as a result of the CSR request. Suppose the KNX IoT device cannot immediately respond due to the time required to generate a key pair; the KNX IoT device SHALL return response code "5.03 Service Unavailable" but uses the Max-Age option to indicate the number of seconds after which to retry.

Constrained KNX IoT devices MAY not have enough power and entropy sources to generate a random private key. In this case, a tool can generate a key pair and write the private key and the LDevID certificate to the device ("auth/skg"). The protocol design for simple enrollment and server-side key generation reuses the same payload format defined in EST-CoAPS [RFC9148] clause 5.3.

The KNX IoT device SHALL use for (D)TLS server authentication the LDevID for requests on "/.well-known/knx/idevid" and the IDevID if the KNX IoT device is in default configuration state. For requests on "/.well-known/knx/LDevID", the KNX IoT device SHALL return a response code of "4.04 Not Found" if the device has no LDevID.

Table 45 specifies the MANDATORY ("M") or OPTIONAL ("O") LDevID resources and the corresponding resource path names on a KNX IoT device:

Table 45 – Security configuration resources

Resource path	rt & Data Types	Format	Method	Support	Request/Response	Notes
auth	:fb.auth	link-format	GET	**M**	Content-Format: application/link-format Payload: </auth/at>;ct=40 60,</auth/crts>;ct=40 281,</auth/sen>;ct=281 286 287,</a/sen>; ct=60,</auth/o/replwdo> ;ct=60,</auth/o/osndela y>;ct=60	The response contains a list with points that belongs to the auth *Functional Block*, incl. the device trust list and access control list (see Table 46, Table 48, Table 45).
/.well-known/knx/ idevid	:dpt.x509	pkcs7-mime; smime-type=ce rts-only	GET	O	Res: Content-Format: application/pkcs7-mime; smime-type=certs-only Payload: <X.509 binary DER encoded>	X.509 binary DER encoded manufacturer device certificate. The server SHALL use the IDevID for (D)TLS sessions in the default configuration state (server certificate). If the server already has a configured LDevID (operational device certificate), then the server SHALL use the LDevID for (D)TLS sessions (server certificate). Resource path is MANDATORY if the device supports (D)TLS.

Resource path	rt & Data Types	Format	Method	Support	Request/Response	Notes
/.well-known/knx/ldevid	:dpt.x509	pkcs7-mime; smime-type=certs-only	GET	O	Res: Content-Format: application/pkcs7-mime; smime-type=certs-only Payload: { X.509 binary DER encoded }	X.509 binary DER encoded operational device certificate. If supported, then the server SHALL use the IDevID (manufacturer device certificate) for the (D)TLS session (server certificate). Resource path is MANDATORY if the device supports (D)TLS.
a/sen	NA	cbor	POST	O	Res: Content-Format: application/cbor Payload: { 2: "renew" }	Triggers LDevID enrollment. MANDATORY if EST enrollment is supported.
auth/sen	:dpt.varOctet	pkcs10	GET	O	Res: Content-Format: application/multipart-core Payload: <PKCS#10>	The enrollment (sen) request based on [RFC9148] (see example clause A.2) is used to read a [PKCS#10] (including proof-of-possession) certificate signing request (CSR) from a KNX IoT device. The request SHALL generate a response code "4.04 Not Found" if the device supports only server-side key generation (skg).
auth/sen	:dpt.varOctet	pkcs7-mime; smime-type=certs-only, pkix-cert	POST	O	Req: Content-Format: application/multipart-core Payload: <PKCS#7> OR Req: Content-Format: application/multipart-core Payload: <Single X.509 binary DER encoded>	The request based on [RFC9148] (see example clause A.2) is used to write a device certificate signing response (PKCS#7) to a KNX IoT device. The request SHALL generate a response code"4.04 Not Found" if the device supports only server side key generation (skg).

Resource path	rt & Data Types	Format	Method	Support	Request/Response	Notes
auth/skg	:dpt.varOctet	/pkcs7-mime; smime-type=100 onfir-generated-key	POST	O	Req: Content-Format: application/pkcs7-mime; smime-type=server-generated-key Payload: { PKCS#7, PKCS#8 }	With server-side key generation, a tool can write the private key and a certificate to the device ([PKCS#7] and [PKCS#8] format). The request based on [RFC9148] (see example clause C.3) SHALL generate a response code "4.04 Not Found" if the device supports only simple enrollment (sen).

3.3 Device Identity Certificates

3.3.1 Manufacturer Device Certificates (IDevID)

The IDevID is used for bootstrapping when the new KNX IoT device is not initialized. The IDevID [802.1AR] manufacturer device certificate is an assertion of the KNX IoT device manufacturer as to the device's unique identity. The IDevID subject field SHALL contain the "serialNumber" attribute with the device's unique *KNX serial number* ([802.1AR] clause 4.1.2.4).

It is RECOMMENDED that the manufacturer device certificate is permanently stored in a security module. The IDevID is only used for commissioning until certificate bootstrapping is completed and gets replaced by the LDevID for normal runtime communication. Hence, the IDevID SHOULD NOT be used for proof-of-identity after successful LDevID certificate bootstrapping but SHALL be reactivated after a device reset to a default configuration state. KNX IoT devices MAY use the IDevID to create different identities for other purposes, such as OSCORE access token configuration in smaller and non-critical installations if no LDevID has been configured.

IDevIDs SHALL have indefinite expiration dates; hence SHALL use the GeneralizedTime value [X.509]. Validating entities SHOULD ignore validity period information in the [X.509] certificate. This ensures that KNX IoT device authentication can always be verified during [X.509] path validation. The Registrar and its network administrator are responsible for determining the trustworthiness of an IDevID and its signer. A manufacturer MAY include a URL that points to a certificate revocation list (CRL) in the signing certificate for the device. A local network (Registrar) SHOULD check the CRL (e.g., manufacturer device CRL published in the Internet or local configuration) before registering a device.

The IDevID X.509 certificate SHALL contain the MANDATORY fields as specified in [802.1AR], including:

- tbs certificate (To Be Signed)
    - subjectPublicKeyInfo
    - issuer; O=[Manufacturer] CN=[Manufacturer CA]
    - subject; O=[Vendor] CN=[Product ID] serialNumber=[*KNX serial number*]
    - validity; Not After: Dec 31 23:59:59 9999 GMT (GeneralizedTime [X.509])
    - version; SHALL be version 3
    - serialNumber; certificate serial number
    - signature
- signatureAlgorithm; SHALL be ecdsa-with-SHA256 and secp256r1
- signatureValue

3.3.2 Operational Device Certificates (LDevID)

The LDevID [802.1AR] operational device certificate is a [X.509] certificate for a particular KNX IoT device, especially in large installations, in the network administrator's responsibility. These certificates are used to identify the device to the network. They are also used to establish [RFC7515] tokens, as required, and MAY be used for different purposes (e.g., at the link layer, network layer, or application layer) for application-layer authentication.

An LDevID contains authentication information in the Subject Alternative Name (SAN) to associate it with a particular security zone. Hence [X.509] certificates SHALL NOT contain multiple security zones (e.g., more than one rfc822Name field). In addition, the wildcard character "*") is neither allowed in the Subject Alternative Name extension nor the common name. The LDevID subject field SHALL contain the "serialNumber" attribute from the IDevID with the device's unique *KNX serial number*. This serial number SHOULD be printed on the device housing (e.g., in a QR code) for visual validation and can be discovered, for example, with /.well-known/core?ep=knx://sn.1caffe1234 or with DNS-SD (see clause 2.6.1 "Discovery").

How an operational device certificate is issued depends on the certificate bootstrapping method and the certificate lifecycle management during operation. The KNX IoT device SHALL have a new key pair before requesting a new operational device certificate. The KNX IoT device either generates the key pair locally or, if possible, obtains the key pair from a trusted Registrar or MaC.

All stored operational device certificates and their associated private keys SHALL be deleted on the device reset to the default state of the KNX IoT device. LDevIDs are expected to be periodically updated. The length of time a certificate is valid is up to the deployment. A RECOMMENDED certificate lifetime is less than one year. However, if the deployment does not have mature processes to update certificates, the lifetime MAY be infinite. Vice versa, if the deployment does not have a mature process to manage the revocation of certificates, then the lifetime MAY be significantly less than one year (e.g., 90 days). Operational certificate Trust Anchors MAY contain a pointer to a CRL. However, KNX IoT devices are not expected to support OCSP.

The LDevID X.509 certificate SHALL contain the MANDATORY fields as specified in [X.509] clause 4.1, including:

- tbs certificate (To Be Signed)
    - subjectPublicKeyInfo
    - issuer;
    - subject; O=[Vendor] CN=[Product ID] serialNumber=[*KNX serial number*]
    - validity
    - version; SHALL be version 3
    - serialNumber; certificate serial number
    - signature
- signatureAlgorithm; see clause 3.4.2, "Device Certificate Cipher Suites"

- signatureValue

3.4 Certificate Validation

3.4.1 General Requirements

KNX IoT devices SHALL follow the procedure defined in RFC 5280 [X.509] to verify certificates. As a summary of the procedure defined in RFC5280:

- Certificate verifiers SHALL reject certificates that contain one or more unsupported critical extensions.
- Any extension not listed by name within this document SHOULD NOT be included within a compliant certificate and, if included, SHALL NOT be marked critical.
- In an authentication exchange, the KNX IoT device SHOULD supply a complete and valid chain comprising its certificate and any intermediate CA certificate between the KNX IoT device and the Root CA.

KNX IoT devices use [X.509] certificates for mutual authentication for device-to-device communication based on (D)TLS. Server identities SHALL be checked as described in [RFC2818] clause 3.1. If a security zone is configured, then the client and server side SHALL check the [X.509] subjectAltName (e.g., rfc822Name) field in the LDevID with the settings in the KNX IoT device Access Control List.

A certificate policy applies to the certification process, from root down to KNX IoT device. This is reflected in the certificate by including the policy OID in all the certificates in the chain. Hence, policy mapping is not supported, and certificates containing policy mappings SHALL be rejected. Therefore, issuers SHOULD NOT include policy qualifiers in operational device certificates. However, verifiers SHOULD accept certificates containing policy qualifiers unless there are other reasons to do so.

To defend against brute-force attacks, the KNX IoT device SHALL limit (D)TLS session establishment retries after a predefined number of consecutive invalid access attempts, e.g., wrong certificate. The KNX IoT device SHALL block further attempts. (D)TLS session establishment attempts SHOULD be limited to at most one attempt per 3000 seconds on average in 24 hours.

3.4.2 Device Certificate Cipher Suites

Elliptic curve cryptography (ECC [RFC4492]) provides the cryptographic basis for secure communication with resource-constrained KNX IoT devices due to its small key size and comparably low arithmetic requirements. All KNX IoT devices supporting identity certificates [X.509] SHALL support the following cipher suite for communication: TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8 [RFC7251].

Specifically, the device identity certificate [X.509] will use ECC NIST P-256 (secp256r1 or prime256v1) [RFC4492]. The signer of the certificate SHALL also use NIST P-256 (secp256r1) and ecdsa-with-SHA256 (ECDSA with SHA256).

Cryptographic suites SHALL be updated from time to time. As such, future releases of this specification may require more than one algorithm for backward compatibility and improved security. Algorithms may also be deprecated.

3.5 Device Access Control

3.5.1 General Requirements

Complementing the core principles of the defense-in-depth protection strategy and the overarching principles of information security, access management itself has a series of core guiding principles, as follows:

- Categorization and classification: Clearly categorize and value all data (see clause 3.5.3 "Access Scope").
- Least privilege: Provide the least amount of access necessary for a given entity to complete its system role.
- Need to know: Provide access to systems and information only where there is a need for the Recipient to have access.
- Controlled access: Define procedures to monitor, enable and disable access methods and enforce security policy at all access points.

Effectively applying these principles to a KNX IoT System throughout processes, networks, and users will ensure that access-related risks are appropriately controlled, allowing authorized access when required and preventing unauthorized access.

The following clauses describe how initial access control privileges for a KNX IoT device are enrolled during the commissioning phase and how a KNX IoT device grants access to resources by verifying requests against the device access control list (configured access tokens) and trust list.

3.5.2 Trust List Resource (auth/crts)

3.5.2.1 General Requirements

A trust list is a list of device certificates or CA certificates that are trusted for authentication and are a MANDATORY functionality of KNX IoT devices if the device supports (D)TLS. The KNX IoT device SHALL reject connections from peers whose device certificate or CA certificate is not in the trust list or if the certificate is expired. Entries in the trust list will be added and removed as an explicit administrative action reflecting changes in trust relationships in conjunction with the access control list; in other words, only a MaC with admin ("if.sec") rights can configure the trust list.

A KNX IoT device SHALL support multiple MaC or Registrars with admin rights. Hence, the trust list SHALL support the configuration of at least 3 CA certificates (e.g., MaC, Registrar, and KDC).

A MaC SHALL write certificates to the KNX IoT device trust list. The [PKCS#7] defines a container format that includes just the public certificate or may include an entire certificate chain or a signature of the encapsulated data structure. The [PKCS#7] or X.509 binary DER encoded format SHALL be used to add new trust relations to other peers. A KNX IoT device MAY support only one format.

Table 46 specifies the MANDATORY ("M") or OPTIONAL ("O") Trust List resources and the corresponding resource path names on a KNX IoT device:

Table 46 – Trust List resources

Resource path	Resource Types (rt)	Format	Method	Support	Request-/Response	Notes
auth/crts	NA	application/link-format	GET	O	Content-Format: application/link-format Payload: </auth/crts/cert-id>;ct=281 287	Collection of configured X.509 certificate thumbprints (cert-id).

Resource path	Resource Types (rt)	Format	Method	Support	Request-/Response	Notes
auth/crts	NA	pkcs7-mime, pkix-cert	POST	O	Req: Content-Format: application/pkcs7-mime; smime-type=certs-only Payload: <PKCS#7> OR Req: Content-Format: application/pkix-cert Payload: <X.509 binary DER encoded>	Configures a PKCS#7 or X.509 binary DER encoded certificate list based on [RFC9148]
auth/crts/{cert-id}	NA	NA	DELETE	O	Res: 2.02 DELETED	Delete a configured token from a KNX IoT device

3.5.3 Access Scope

Access Scopes are the permission grants that client applications require for authorization and use of Points on a server device. A Point SHALL be assigned to at least one access scope. The access rights of a client are limited to the granted scopes in the access token or the device access control list (e.g., for OSCORE).

The number of scope identifiers (the granularity) supported by the server device is device-specific. Generally, it matches the capabilities of the server device and the expected kinds of data it contains. However, KNX IoT devices (servers) SHALL support the predefined scope meanings and identifiers, and for interoperability, it is RECOMMENDED to use the predefined scope identifiers whenever possible.

The following table defines the default KNX IoT access scopes. Scopes are mapped to one or more interfaces (if). The interface definition and the allowed CoAP Methods for a particular scope are defined in clause 2.5.3, "Interface Types (if)":

Table 47 – KNX IoT Access scopes

Scope	CBOR Type	if	Description
if.i	text string	if.i	Write and command runtime input Group Object Point.
if.o	text string	if.o	Read and subscribe runtime output Group Object Point.
if.g.s	text string	if.g.s	Group communication (S-Mode) runtime interworking (input and output) for all *Group Addresses* ("ga").
<ga>	unsigned	if.g.s	Group communication (S-Mode) runtime interworking (input and output) for a particular *Group Address* ("ga").
if.p	text string	if.p, if.ll	Adjust (write, read) Parameter Property (Point), incl. metadata.
if.d	text string	if.d, if.ll	Read Diagnostic Property (Point), incl. metadata.
if.a	text string	if.a	Hardwired actuator (digital or analog output).

Scope	CBOR Type	if	Description
if.s	text string	if.s	Hardwired sensor (digital or analog input).
if.c	text string	if.p, if.d, if.ll	Configure device (commissioning) functionality or Function Point Tables.
if.sec	text string	if.sec, if.swu	Configuration (read and write) of security, incl. metadata and authorization-related data.
if.swu	text string	if.swu	Software update (push and pull) related data, incl metadata.

3.5.4 Device Access Control List Resource (auth/at)

3.5.4.1 Common requirements

The device access control list is a MANDATORY functionality of KNX IoT devices and specifies which entities (user, application, or device) have access. The list contains permissions, which control access to KNX IoT device resources (Points) and what operations are allowed on given resources (GET, PUT, etc.).

Each permission in the device access control list identifies an authorized entity and specifies the access rights for that entity (e.g., allowed or denied). It is up to the system integrator, in collaboration with the network administrator, to define permission rules for an entity. As a precondition for any access control decisions, the KNX IoT device SHALL authenticate and authorize the peer by verifying its [X.509] certificate and/or pre-shared key (OSCORE). Verification includes the following steps:

- The pre-shared key SHALL be configured in the device access control list (access token).
- The pre-shared key SHALL have permission rights on the resource server to access the corresponding resource (access scope).
- The operational device certificate is correctly signed by a trusted CA (see clause 3.4). The trusted CA certificate SHALL be configured in the device trust list.
- The peer's identity in the [X.509] operational device certificate (e.g., Subject and/or SubjectAltName name) is an entity that is authorized to access the required resources. The entity SHALL be validated with the corresponding access token (access scope) in the access control list.

Suppose a security zone has no exclusive CA certificate, such as an intermediate CA. In this case, the KNX IoT device access token SHALL contain settings in the subject (sub) attribute equivalent to name constraints defined in [X.509] clause 4.2.1.10.

Devices that cannot reliably keep track of time SHALL accept all credentials even if the credential lifetime has expired. This requirement MAY change in a future specification, but no expiration time can be configured yet for pre-shared key access tokens (OSCORE). It is the responsibility of the KDC to rotate and delete access tokens with an appropriate interval and the overlap of active and new pre-shared keys.

Group communication credential configurations SHALL contain only *Group Address* <ga> scopes in the access token. A combination with other scopes, such as "if.p", is impossible since incoming and outgoing communications use the same credentials. For example, devices in the following figure with the access token ID 211, 212, 221, and 241 SHALL accept incoming and outgoing messages on configured *Group Addresses*.

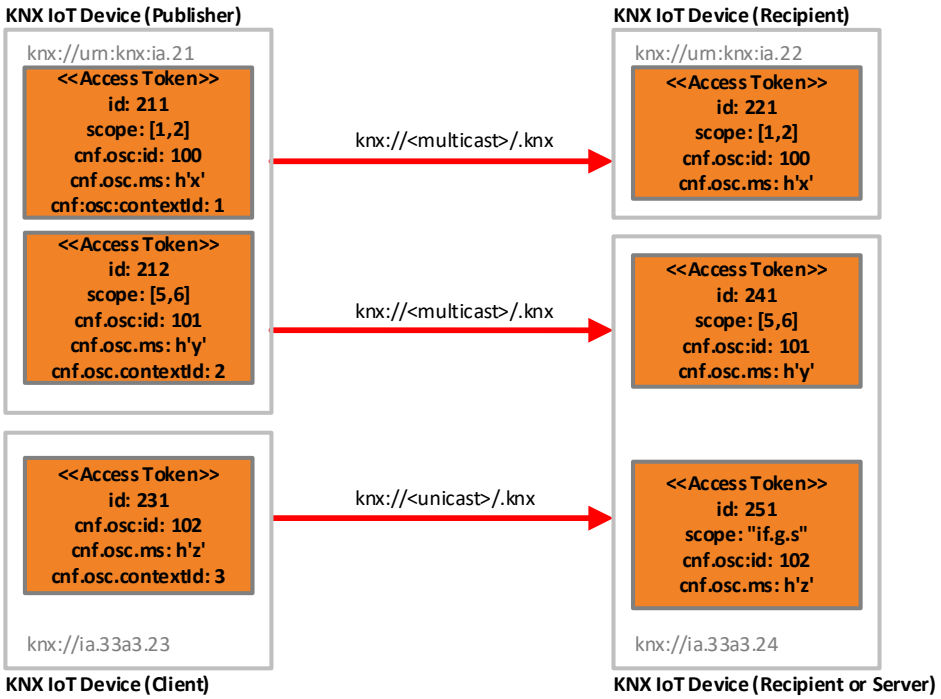


Figure 19 – Access Token Configuration Example

The MaC SHALL configure the corresponding access token in the Function Point Table for outgoing messages (Publisher or client KNX IoT device). Figure 19 depicts two KNX IoT devices (Publisher) that send multicast and unicast messages to the devices with the *KNX Individual Address* "24" and "22". The corresponding access tokens are configured in the Function Point Recipient Table.

The MaC can limit permissions on the resource server for an access token to a set of *Group Addresses*. The multicast examples in Figure 19 have a limited set of *Group Addresses* configured in the access scope on the Recipient device. Only S-Mode messages with the corresponding *Group Addresses* (1/2 or 5/6) are accepted. However, the Publisher device SHALL ignore scope definitions in the access token.

A MaC MAY configure the same OSCORE credentials for device subscriptions on multiple devices. A typical use case for such a configuration is a controller with multiple sensors and actuators. Figure 20 depicts a KNX IoT device that subscribes (Subscriber/client) to "/.knx", and the publishing device (Publisher/server) sends notification messages back to the Subscriber. The Subscriber KNX IoT device has the corresponding access tokens configuration in the Function Point Publisher Table. However, the Publisher device SHALL only send S-Mode notifications containing the configured *Group Addresses* in the scope of the access token that was used for the corresponding subscription request.

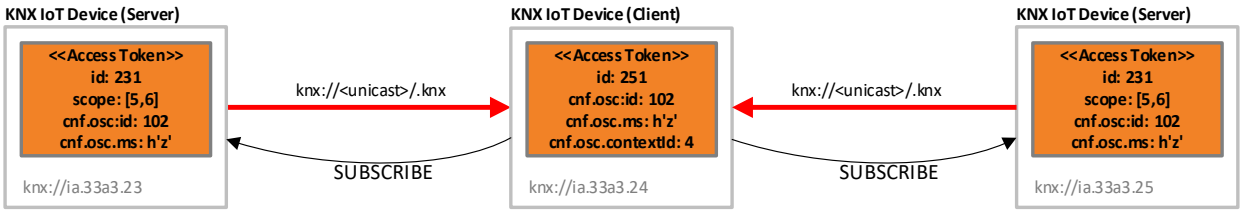


Figure 20 – Subscription Access Token Configuration Example

Suppose the MaC wants to change access token credentials (e.g., pre-shared tool key) on a KNX IoT device. In this case, the MaC SHALL use different credentials for writing than those credentials configured in the new modified access token (existing token to be changed). Therefore, the MaC SHALL configure the new access token first and delete the old access token with a subsequent request.

2394 Table 48 specifies an example of the MANDATORY ("M") or OPTIONAL ("O") token configuration
2395 resources on a KNX IoT device.

2396 **Table 48 – Access token configuration resources on a KNX IoT device (EXAMPLE)**

Resource path	Resource Types (rt)	Format	Method	Support	Request/Response	Notes

auth/at	NA	cbor	POST	**M**	Req: Content-Format: application/cbor
					Payload:
					{
					0:"oc5",
					9: ["if.sec"],
					38: 2,
					8: { 4: {
					4: 10,
					0: <id-osc>,
					2: <ms>
					}}}
						Credentials configuration for KNX IoT device access. For example, a device (Registrar or KDC) can configure access token profiles for security credentials ("if.sec") with its X.509 certificate or existing OSCORE token.
						An empty element (only "id") in the payload will cause the deletion of a corresponding existing element.

auth/at	NA	application/link-format	GET	**M**	Content-Format: application/link-format Payload: </auth/at/12345>;ct=60	Reads a collection of all configured tokens from a KNX IoT device

auth/at/{token-id}	NA	cbor	GET	**M**	Res: 2.05 CONTENT
					Content-Format: application/cbor
					Payload:
					{
					0: "bc6",
					2: "<.inst1.local>",
					9: ["if.sec"],
					38: 254,
					8: {
					4: "<trust list [X.509]fingerprint>"
					}}
						Reads a specific token profile from a KNX IoT device (e.g., coap_oscore or coap_tls with X.509 certificates),

auth/at	NA	NA	DELETE	**M**	Res: 2.02 DELETED	Delete all configured tokens from a KNX IoT device

Resource path	Resource Types (rt)	Format	Method	Support	Request/Response	Notes
auth/at/{token-id}	NA	NA	DELETE	**M**	Res: 2.02 DELETED	Delete a configured token from a KNX IoT device

2397

2398 ### 3.5.4.2 Access Token Resource Object

2399 For KNX IoT access rights configuration, the CWT (CBOR Web Token) format [RFC8392] is used in
2400 combination with [RFC8747]. The following fields are MANDATORY for each access token item:

2401 • id: Token ID for updating and deleting an item on a device. The ID SHALL contain only
2402 characters as defined for the URI-Reference in [RFC3986] since the ID is also used in the access
2403 token resource path.
2404 • cnf: Key configuration (either a link to the trust list or OSCORE key settings).
2405 • profile: The token profile defines further MANDATORY members (M).
2406 • scope: Needed for both X.509 certificate and OSCORE.

2407 Table 49 specifies the MANDATORY ("M") or OPTIONAL ("O") access token attributes and names.
2408 MANDATORY ("M") attributes SHALL be present in the message, and OPTIONAL ("O") may be
2409 present in the message, but the attributes SHALL be supported on the device.

2410 **Table 49 – Access token item members**

JSON Key	CBOR Key	CBOR Type	Support	Name
"id"	0	text string	**M**	Access token ID as ASCII string of numbers and lowercase letters (maximum length: 32 bytes); The ID SHALL be unique on a device and SHOULD be unique in an installation.
"cnf"	8	map	**M**	confirmation key; e.g., with a symmetric proof-of-possession key [RFC 8392]
"osc"	4	map	**(M)**	OSCORE confirmation key (see clause 3.6.2 "OSCORE Key Configuration Resource Object")
"kid"	2	text string	O	Key Identifier: Acts as an "alias" for the key and MAY be a Link to [X.509] certificate in the Trust List (e.g., x.509 thumbprint). Note: This "kid" is used for X.509 certificates but not for the "kid" in the OSCORE message.
"sub"	2	text string	O	Subject of the access token (used if [X.509] CA cert. is ambiguous, and the SAN SHALL be restricted) [RFC 8392]
"exp"	4	unsigned	O	The token is valid until a defined time. Epoch-based date/time (NumericDate [RFC8392])

JSON Key	CBOR Key	CBOR Type	Support	Name
"nbf"	5	unsigned	O	Token is not valid before a defined time (Not before). Epoch-based date/time (NumericDate [RFC8392])
"scope"	9	string array, unsigned array	**M**	Array of interfaces as defined in clause 3.5.3 "Access Scope" [RFC 8392] (string) or a list of *Group Addresses* (unsigned). The Access Control List SHALL support >= 20 array elements.
"profile"	38	unsigned	**M**	access token profile; SHALL be set to 1: "coap_dtls" for [X.509] certificates with DTLS [RFC 8392], 2: "coap_oscore" [OSCORE], 254: "coap_tls" [OSCORE] for [X.509] certificates with TLS, or 255: "coap_pase" [OSCORE] with PASE credentials

3.5.5 Revocation List

A revocation list is a Blacklist of digital certificates that a trustworthy entity has revoked before their scheduled expiration date. Digital certificates (X.509 certificates) in the revocation list are no longer trusted, and requests SHALL be declined from devices that use a certificate from the revocation list. Digital certificates are revoked for many reasons. For example, if a certificate is discovered to be counterfeit, the Trust Anchor will revoke it and add it to the revocation list of each KNX IoT device in the system. A common reason for revocation occurs when the owner of the Domain CA no longer owns devices, or the original certificate is replaced with a different certificate from a new Trust Anchor (e.g., after commissioning).

The problem with revocation lists, as with all Blacklists, is that they are difficult to maintain and are an inefficient and unreliable method of distributing in real-time. Furthermore, the revocation list size is limited, especially on constrained devices; therefore, a revocation list on a KNX IoT device is RECOMMENDED. If present, the list SHOULD be updated only under exceptional circumstances. The process of updating revocation lists is manufacturer-specific and not part of this document.

3.6 OSCORE Application Layer Security

3.6.1 General Requirements

This clause describes the configuration of OSCORE credentials. OSCORE application layer security is needed for the following reasons:

- Low-latency communication: (D)TLS can only be used for unicast point-to-point communication but not for secure multicast group communication (e.g., switch on a light group). In addition, (D)TLS needs a time-consuming (on constrained devices) session key establishment before peers can start communicating. The resulting latency is not acceptable for some applications like lighting.
- Application layer end-to-end security: The (D)TLS security terminates at proxies and message brokers, making this intermediary node capable of accessing and manipulating any part of the message payload and metadata.

OSCORE, as defined in this document, uses pre-shared keys to secure communication between peers (no session establishment needed). OSCORE protects unicast and multicast CoAP application payload end-to-end across intermediary nodes (i.e., proxies and message brokers). Therefore, all KNX IoT devices SHALL support OSCORE for unicast and multicast communication.

OSCORE [RFC8613] essentially protects the RESTful interactions (including replay protection), the request method, the requested resource, the message payload, etc. OSCORE neither protects the CoAP messaging layer nor the CoAP token, which may change between the resources. OSCORE requires all KNX IoT devices belonging to the same KNX group to establish a shared security context used to process COSE [RFC8152] (CBOR Object Signing and Encryption) objects. OSCORE uses COSE with an AEAD [RFC5116] (Authenticated Encryption with Additional Data) algorithm for protecting message data (encryption) between a client and a server. The CoAP payload SHALL be encrypted in the COSE object.

The following figure depicts the general workflow for configuring access tokens on a KNX IoT device. The access token configuration flow includes the following three steps:

- The MaC has a means to authenticate the KNX IoT device. This MAY be an authentication code or an LDevID after device identity enrollment (see clause 3.2). If the KNX IoT device already has an LDevID, then the Registrar (e.g., MaC) SHALL have "if.sec" access rights configured in the access control list of the KNX IoT device.
- Suppose the MaC has just an authentication code; then the MaC uses the Password Authenticated Session Establishment (PASE) for device authentication (see clause 3.6.3). Subsequently, the MaC uses a trusted PASE session for reading the IDevID of the KNX IoT device.
- If supported by both peers, then the MaC starts a (D)TLS session and uses the IDevID for authentication, and configures OSCORE Access Tokens for the tool key ("if.sec") and runtime keys ("if.s", "if.p", "if.d", etc.).

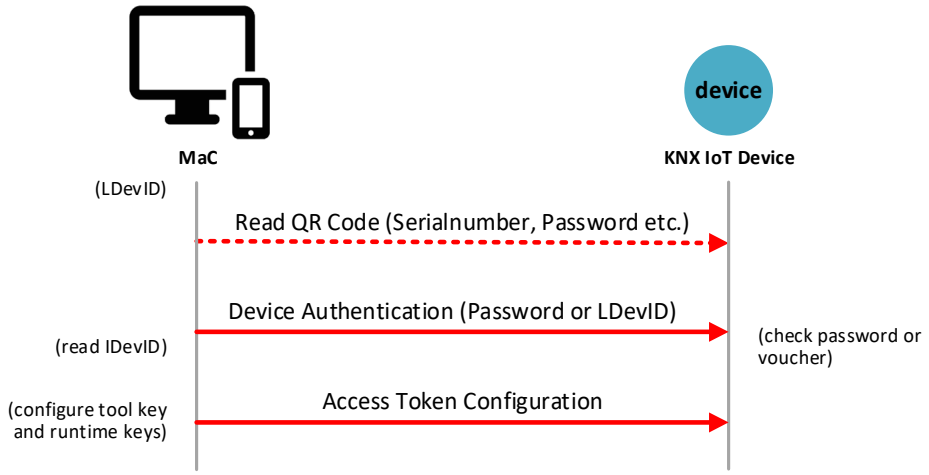


Figure 21 – OSCORE Access Token Configuration

3.6.2 OSCORE Key Configuration Resource Object

This clause defines how to derive a security context based on a shared master secret and a set of other parameters established between all KNX IoT devices that exchange data. The derivation of the Sender Key, Recipient Key, and Common IV SHALL be done according to [RFC8613] clause 3.2. In the case of pre-shared keys, the proof-of-possession key (runtime key), provisioned from the MaC MAY, is used directly as a master secret in OSCORE. If OSCORE is used directly with the symmetric proof-of-possession keys as the master secret, then the MaC SHALL provision the master secret ("ms"), the context ID, and the client ID. Additionally, the MaC MAY provision the "hkdf", "alg", or "salt" parameter. If these parameters are omitted, the default values are used as defined in [RFC8613] clause 3.2.

For KNX IoT OSCORE access rights configuration, the CWT format, based on [RFC9203], contains the necessary parameters in the "cnf" claim. Table 50 specifies the MANDATORY ("M") or OPTIONAL ("O") attributes and defines the CBOR mapping for the OSCORE_Security_Context. MANDATORY ("M") attributes SHALL be present in the message, and OPTIONAL ("O") may be present in the message, but the attributes SHALL be supported on the device

Table 50 – Members in the "cnf"-claim

JSON Key	CBOR Key	CBOR Type	Support	Description
"id"	0	byte string	**M**	OSCORE input material identifier (osc-id maximum length: 7 bytes). The id SHALL NOT be an empty byte string (h").
"version"	1	unsigned	O	OSCORE Version.
"ms"	2	byte string	**M**	master secret; SHALL be the PSK (32 bytes).
"hkdf"	3	unsigned	O	KDF (Key Derivation Function algorithm); Default SHALL be HKDF SHA-256 [RFC5869].
"alg"	4	unsigned	O	AEAD algorithm; SHALL be set to the value from [RFC8152]. The default is AES-CCM-16-64-128 (COSE algorithm encoding: 10).
"salt"	5	byte string	O	salt; Default SHALL be an empty byte string.
"contextId"	6	byte string	**(M)**	The context ID SHALL be used as "kid_context" in the OSCORE message (maximum length: 3 bytes); The context ID is used to distinguish security contexts (source device) on the receiver. The context ID is MANDATORY on the sending device but SHALL be ignored or omitted in the access token on the receiving device.

The MaC SHALL send a POST request to "auth/at" to configure tokens on a KNX IoT device. The payload SHALL be a CBOR map. The configuration request to the KNX IoT device SHALL be secured either with CoAP over (D)TLS (if supported by the device) or OSCORE. The scope attribute can be omitted if all Points of the assigned IPv6 multicast address ("serverId") use the same access token. However, if a *Group Address* is configured with an individual token, then the scope SHALL be configured with the *Group Address* as defined in clause 2.5.9, "S-Mode Messaging Resource (.knx)". A Group Message with a configured security context SHALL be protected by OSCORE for multicast and unicast (with and without (D)TLS).

Table 51 specifies MANDATORY ("M") OSCORE resources and the corresponding resource path on a KNX IoT device.

2490 **Table 51 – OSCORE configuration resources**

Resource path	Resource Types (rt)	Format	Method	Support	Request/Response	Notes

auth/at	NA	cbor	POST	**M**	Content-Format: application/cbor
					Payload:
					{
					0: "oc7",
					38: 2,
					9: [<ga>],
					8: {
					4: {
					4: 10,
					0: "<osc-id>",
					2: "<ms>"
					}}}
						OSCORE multicast credentials config. For KNX Group communication.

					Content-Format: application/cbor
					Payload:
					{
					0: "oc5bhk",
					38: 2,
					9: [<ga>],
					8: {
					4: {
					4: 10,
					0: "<osc-id>",
					2: "<ms>"
					}}}
						OSCORE unicast credentials config. For KNX Group Communication. However, it is also possible to configure a token for a Point subscription ("scope": ["<PointInterface>"])

2491

2492 3.6.3 Password Authenticated Access Token Enrollment

2493 3.6.3.1 Introduction

2494 This clause describes an access token configuration procedure using a shared password as an
2495 authentication code for devices in the default configuration state. At the end of the procedure, the KNX
2496 IoT device has received credentials (tool key="if.sec") allowing for further access tokens configuration.

2497 PAKE protocols allow two or more entities to authenticate each other and create an encrypted channel for
2498 further communication. The security protocols rely on an out-of-band secret (e.g., password from a
2499 printed QR code or retrieved via NFC interface from KNX IoT device) to authenticate the exchanges. The
2500 password is never sent, but parties in the exchange can prove possession of the shared password and thus
2501 authenticate. When both parties have computed the same PASE session key, it is used to symmetrically
2502 encrypt all data in subsequent OSCORE messages of the same PASE session.

SPAKE2+ [13] is an augmented Password Authenticated Key Exchange (PAKE) protocol where only one party directly uses the password. The other party may not know the password but only stores a derivative of the password. The party using the password directly typically is the initiator/client (e.g., MaC) and acts as a prover. In contrast, the other party is a server and acts as a verifier (KNX IoT device).

SPAKE2+ is agnostic to the used transport protocol (e.g., Bluetooth, TCP, etc.). The following clause describes how CoAP is used as the SPAKE2+ transport protocol.

3.6.3.2 PASE Resource Object

PAKE key enrollment uses CoAP [RFC7252] as a reliable transport protocol and OSCORE [RFC8613] for encryption. It is RECOMMENDED to carry messages in confirmed messages, mainly if fragmentation is used. Table 52 defines JSON keys and the CBOR mapping for CoAP request and response objects to "/.well-known/knx/spake".

Table 52 defines the MANDATORY ("M") attributes the client and the server SHALL support.

Table 52 – JSON keys and the CBOR mapping

JSON Key	CBOR Key	CBOR Type	Support	Description
"id"	0	text string	**M**	Responder ID context as ASCII string of numbers and lowercase letters (maximum osc-id length: 7 bytes). The id SHALL NOT be an empty text string ("").
"salt"	5	byte string	**M**	salt: A random value of at least 16 bytes and at most 32 bytes.
"shareP"	10	byte string	**M**	shareP: The SPAKE2+ contribution in uncompressed public key format 32 bytes).
"shareV"	11	byte string	**M**	shareV: The SPAKE2+ contribution in uncompressed public key format.
"pbkdf2"	12	map	**M**	*PBKDF2*: Password-Based Key Derivation Function 2
"confirmV"	13	byte string	**M**	confirmV: The key confirmation.
"confirmP"	14	byte string	**M**	confirmP: The key confirmation.
"rnd"	15	byte string	**M**	Random value (32 byte)
"it"	16	unsigned	**M**	PBKDF Iterations: An integer value specifying the number of iterations (see clause 3.6.6.4, "SPAKE2+ Enrollment Configurations").

3.6.3.3 Device Authentication with SPAKE2+ over CoAP

The following figure depicts an Initiator acting as a CoAP client and the Responder acting as a CoAP server. The Password Authenticated Session Establishment (PASE) is used for generating the OSCORE master secret. Every KNX IoT device SHALL have a unique password that is used as proof of possession in the PASE flow (see clause 3.6.6.6 "Password").

The Initiator (e.g., MaC) initiates the PASE Parameter exchange with the Responder (KNX IoT device). In the response, the Responder sends an unencrypted salt to the verifier. This salt and the KNX IoT device password are used from both peers as input for the key derivation function (see clause 3.6.6.2, "Password-based Key Derivation Function (PBKDF)").

The CoAP client SHALL send a POST request containing the SPAKE2+ [13] message 1 to a reserved resource at the CoAP server (PASE Credential Request). If the KNX IoT device is in the default configuration state, this message triggers the SPAKE2+ exchange on the CoAP server, which SHALL reply with a "2.04 Changed" response containing the SPAKE2+ message 2 (PASE Credential Response).

Finally, the SPAKE2+ message 3 (PASE Credential Verification) is sent by the CoAP client in a CoAP POST request to the same resource used for the SPAKE2+ message 1.

The Content-Format of these CoAP messages SHALL be set to "application/cbor".

If the device onboarding with PASE fails after a predefined number of consecutive invalid access attempts, e.g., wrong password, then the KNX IoT device SHALL block further attempts. The server SHALL return response code "5.03 Service Unavailable" in the last response and SHALL use the Max-Age option to indicate the number of seconds after which to retry. It is RECOMMENDED to block device onboarding after ten (10) consecutive invalid attempts per minute. It MAY increase the timeout with every failed retry but no longer than 5 minutes (see clause 3.6.6.6, "Password").

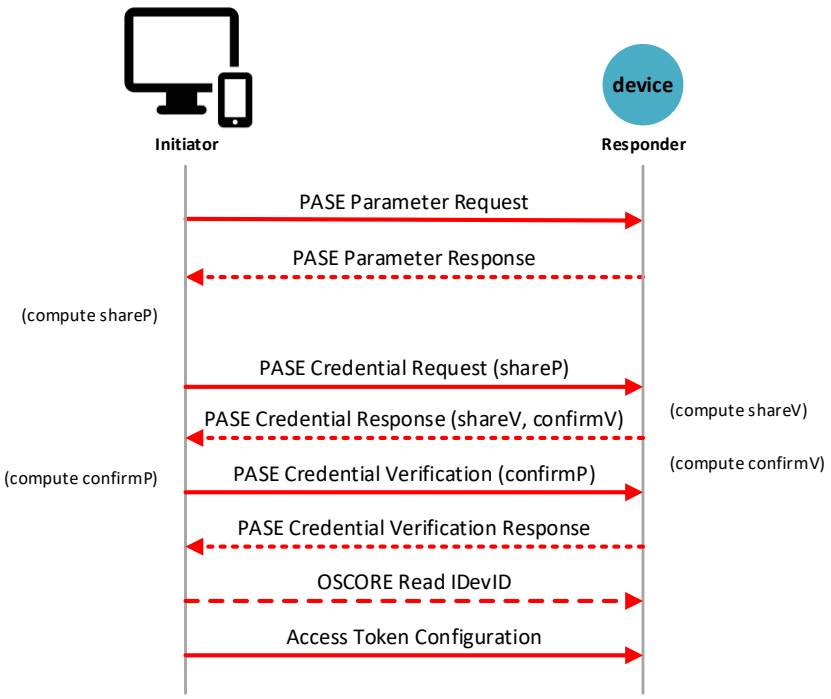


Figure 22 – Password Authenticated OSCORE Access Token Enrollment

The PASE Parameter exchange SHALL contain the following attributes from Table 52 for the request:

- rnd

The PASE Parameter request SHALL contain a responder key ID for the temporary OSCORE access token. The responder key ID in the request SHALL be the OSCORE input material identifier (osc-id) for the following access token configuration.

- id

The PASE Parameter response SHALL contain the following attributes:

- rnd
- pbkdf2
    - it
    - salt

PASE parameter exchange.

REQ:
POST coap://{ipv6-unicast}/.well-known/knx/spake
(Accept: application/cbor (60))
Payload:
{ 0: "rkey", 15: h'3456…c45e' }

RES:
2.04 CHANGED (Content-Format: application/cbor (60))
Payload:
{ 15: h'8dfs…a7ad', 12: { 16: 1000, 5: h'98fa…fw54' } }

With the following message, the Initiator (e.g., MaC) responds with its public key shareV (see clause 3.6.3.3 "Device Authentication with SPAKE2+ over CoAP"). Both Responder (KNX IoT device) and Initiator SHALL derive a shared secret to generate encryption and authentication keys. The Responder SHALL send the verifier (Initiator) a key confirmation message containing confirmV so both parties can confirm that they agree upon these shared secrets.

The PASE Credential request SHALL contain the following attributes from Table 52:

- shareP

The PASE Credential response SHALL contain the following attributes:

- shareV
- confirmV

PASE credential exchange.

REQ:
POST coap://{ipv6-unicast}/.well-known/knx/spake
(Accept: application/cbor (60))
Payload:
{ 10: h'b6b8…95c9' }

RES:
2.04 CHANGED (Content-Format: application/cbor (60))
Payload:
{ 11: h'9876…aa98', 13: h'a987…c3874' }

The Initiator (e.g., MaC) SHALL send to the Responder (KNX IoT device) a confirmation message containing confirmP. The Initiator SHALL NOT send application data to the Responder until it has received and verified the confirmation message. Key confirmation verification requires the recomputation of confirmP or confirmV and checking for equality against that which was received (SPAKE2+ [13] clause 4).

The PASE Credential Verification request SHALL contain the following attributes from Table 52 in the request:

- confirmP

PASE Credential Verification.
REQ: POST coap://{ipv6-unicast}/.well-known/knx/spake (Accept: application/cbor (60)) **Payload:** { 14: h'8964…d887' } **RES:** 2.04 CHANGED

As a result of the PASE credential exchange, each party can compute their sending and receiving keys. The resulting shared symmetric secrete K (Ke according to SPAKE2+ [13] clause 3.5) SHALL be used as master secret ("ms") for operational OSCORE access token configuration.

The following example shows the OSCORE access token configuration after PASE credential exchange. This temporary PASE access token item is generated from the KNX IoT device and SHALL be deleted from the device as soon as the token expires. This access token grants access to the Initiator with the shared secrete K (responder key) from the previous PASE session. Only one PASE responder key is valid at a time.

The access token "id" and the OSCORE input material identifier (osc-id) SHALL be taken from the PASE parameter exchange resource object "id". With this, the MaC can delete the temporary PASE access token with a DELETE request to, for example, "/auth/at/rkey" after configuring a new access token with the tool key.

The Initiator with the PASE responder key can configure additional operational access tokens for a limited time.

A KNX IoT device SHALL stop access token enrollment after a specified period (e.g., 5 min) if the SPAKE2+ session establishment has not been concluded. It is RECOMMENDED that the temporary PASE access token and the corresponding responder key gets invalid after a device-specific timeout.

Table 53 – Access token example after PASE Credential Exchange

Resource path	rt & Data Types	Format	Method	Support	Request/Response	Notes

auth/at/{token-id}	:dpt.swor	cbor	GET	**M**	Content-Format: application/cbor
					Payload:
					{
					0: "rkey",
					38: 255,
					9: ["if.sec"],
					8: {
					4: {
					4: 10,
					0: h'726b6579',
					2: "<Derived shared secret K from PASE>"
					}}}
						The PASE-based access token (profile: coap_pase) is only applicable for "scope": ["if.sec"]. This OSCORE access token with a pre-shared key is used for further access token configurations. In this example, the device has internally added a token to the access control list with "rkey" for the key IDs from the PASE parameter exchange.

3.6.3.4 Access Token Configuration

This clause describes how a MaC or a 3rd Party device configures access tokens on a KNX IoT device during commissioning (PAKE) and after commissioning. The precondition is that the MaC or the 3rd Party device has access to the corresponding credentials in the KDC service (Key Distribution Center).

The key management of a KDC is out-of-scope of this document and is the network administrator's responsibility. However, it is RECOMMENDED to have a Tool Master Key for security configuration (access scope = "if.sec") and a Runtime Master Key. The individual tool and runtime keys are derived from the Master secret (see clause 3.6.6.1 "Key Derivation Function") for each KNX IoT device that the 3rd Party device integrates. The Master Secret SHOULD be accessible to the network administrator. The Master Secret is important in case the 3rd Party gets replaced by a new device. With the Master Secret, the new 3rd Party device can bootstrap all existing KNX IoT devices and configure new runtime keys. The Runtime Master Key MAY be generated device internally and is only known by the 3rd Party device.

If both the 3rd Party device and the KNX IoT device support (D)TLS, then it is RECOMMENDED to configure the LDevID of the 3rd Party device in the access token with "if.sec" access scope. In this case, the Master Secret, or rather the tool key, gets replaced by an X.509 certificate.

3.6.3.4.1 Temporary Access Token Configuration (Responder Key)

The 3rd Party device integrates a KNX IoT device in the following example. The 3rd Party device configures access tokens (incl. pre-shared keys) and, in addition, can read, write and subscribe to application data (e.g., room temperature value) later after commissioning.

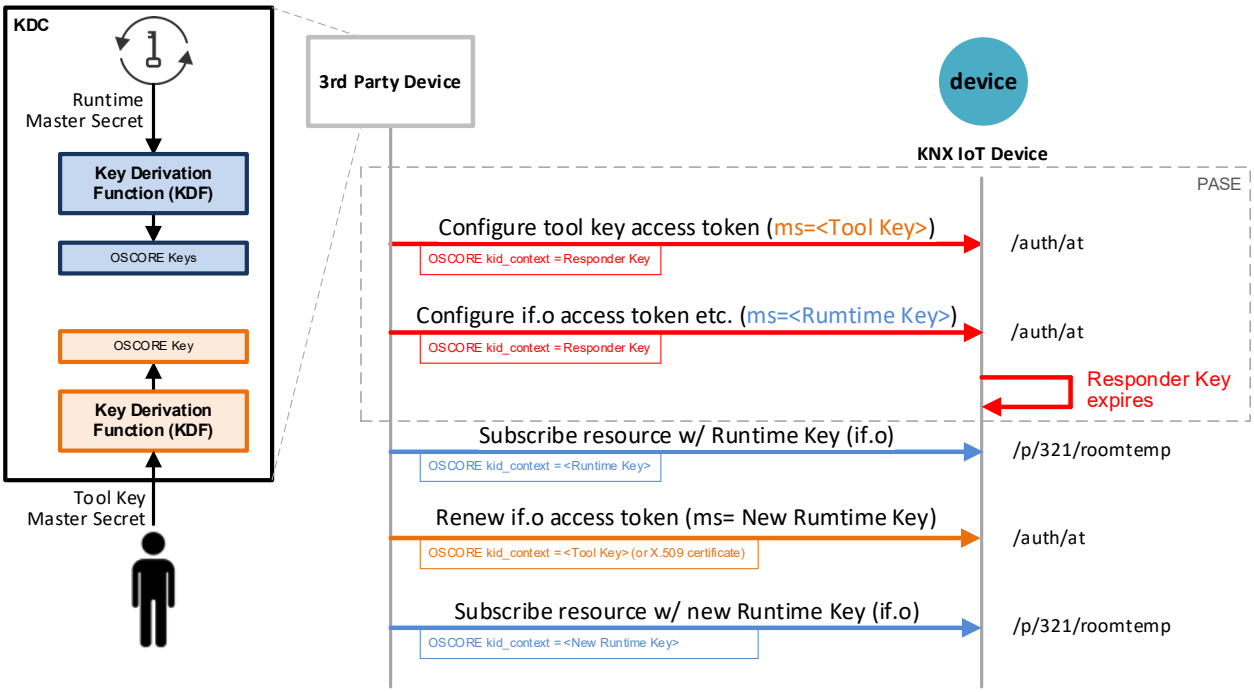


Figure 23 – Access Token Configuration (Pub/Sub)

The sequence diagram above continues with an existing PASE session (valid responder key). The Initiator (3rd Party device) can only configure access tokens since the access token scope of the PASE responder key SHALL be limited to "if.sec".

The following CoAP POST example shows how the Initiator configures a new operational OSCORE access token for the "if.sec" scope (tool key).

The KNX IoT device SHALL accept the first OSCORE unicast request from an unsynchronized client. However, the client SHALL create a randomized sequence number for the first request. The sequence number from the client SHALL be deleted with the PASE security context when the PASE access token expires or gets deleted. This is an exception to the general rule defined in clause 3.6.4, "Message Replay Protection".

The CoAP message SHALL contain the PASE "id" as "kid". Before accepting the request, the Responder (KNX IoT device) SHALL check the corresponding credentials. At least one operational access token configuration SHALL contain the "if.sec" scope; otherwise, access token changes or renewals are impossible as soon as the PASE responder key expires.

The client in the following example uses the kid=h'726b6579'="rkey" (osc-id) and the PASE responder key ms=<derived shared secret K from PASE> to configure a new access token (tool key) containing a reference to an X.509 certificate:

Configure tool key access token (ms=<tool key>) for an X.509 certificate.

```
REQ:
POST coap://{ipv6-unicast}/auth/at
(Content-Format: application/cbor (60), OSCORE(code(POST), kid((h'726b6579')))
Payload:
{
 0: "12345",                        //id
 9: ["if.sec"],                     //scope
 2: "<kdc-id.knx.local>",           //sub
 38: 254,                           //profile: coap_tls
 8:{                                //cnf
 2: "<trust list [X.509]fingerprint>" }      //kid
}
```

The initiator MAY configure additional runtime keys (configure "if.o" access token) with the same PASE responder key session.

The previous example shows an X.509 certificate-based trust to the entity that hosts the KDC. If both peers support (D)TLS, then the Initiator SHOULD configure the "coap_tls" or "coap_dtls" profile for X.509 certificate-based mutual authorization for the tool key (see clause 3.5.4 "Device Access Control List Resource (auth/at)").

However, it is also possible to configure a coap_oscore profile for the tool key. Depending on local security requirements, the local network administrator has to decide whether to use an X.509 certificate or a pre-shared key for the tool key.

The client in the following example uses again the kid=h'726b6579'="rkey" (osc-id) and the PASE responder key ms=<derived shared secret K from PASE> to configure further access tokens with a new <osc-id>:

Configure tool key access token (ms=<tool key>) with an OSCORE pre-shared key.

REQ:
POST coap://{ipv6-unicast}/auth/at
(Content-Format: application/cbor (60), OSCORE(code(POST), kid((h'726b6579')))
Payload:

```
{
 0: "23451a",              //id
 38: 2,                    //profile: coap_oscore
 9: ["if.sec"],            //scope
 8: {                      //cnf
 4: {                      //osc
 4: 10,                    //alg: AES-CCM-16-64-128
 0: "<osc-id>",            //id
 2: "ca5….4dd43e4c"        //ms
}}}
```

3.6.3.4.2 Access Token for Group Communication

The following example shows a MaC configuring access tokens on two KNX IoT devices that want to exchange S-Mode group notifications. Hence, the MaC SHALL configure the same runtime key on all KNX IoT devices that send or receive messages with a particular *Group Address*. Each *Group Address* in the access token SHOULD have a corresponding *Group Address* configuration in the Function Point Table.

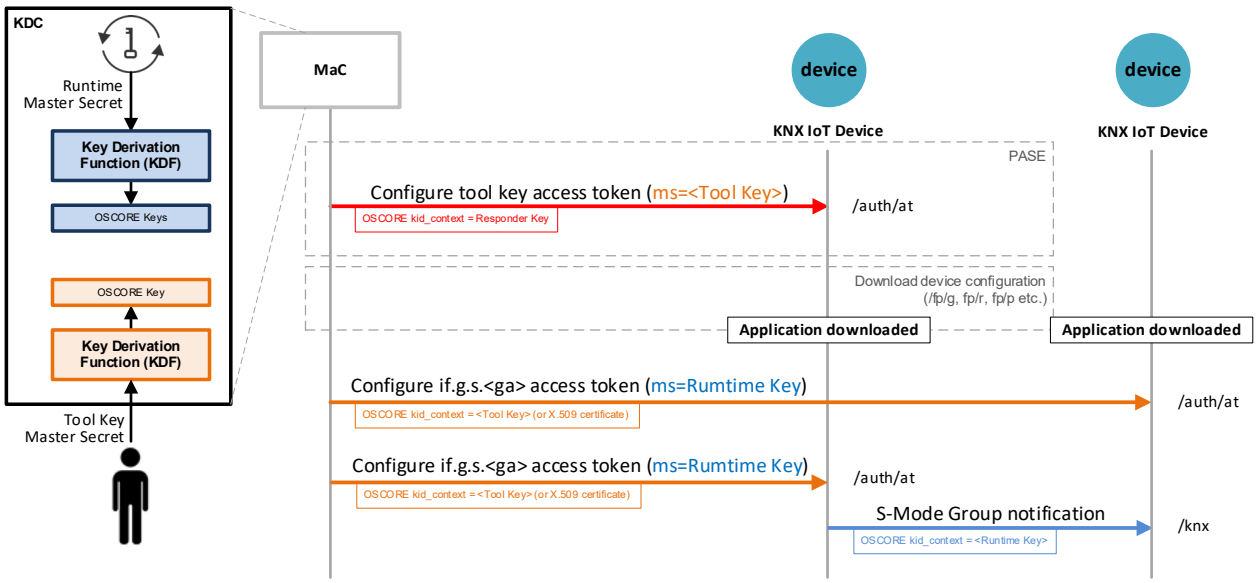


Figure 24 – Access Token Configuration (Group Communication)

An access token for a *Group Address* contains a specific scope identifier, as shown in the following table. It is an extension of the previous example with a 3rd Party device.

If a POST request successfully creates a new access token, the server SHALL return a "2.01 Created" response code. If an access token exists and the update is successful, the server SHALL return a "2.04 Changed" response code with no response document. A valid access token item SHALL contain all MANDATORY members as defined in clause 3.5.4.2. Partial update of access token items SHALL be supported, for example, for key renewal.

The client in the following example uses the newly configured kid and the ms=<tool key> from the previous example to configure further access tokens with a new <osc-id>.

Configure <ga> access token (ms=<runtime key>).

REQ:
POST coap://{ipv6-unicast}/auth/at
(Content-Format: application/cbor (60), OSCORE(code(POST), kid(<osc-id>)))
Payload:

```
{
  0: "345612",                 //id
  38: 2,                       //profile: coap_oscore
  9: [1, 2, 3, 4],             //scope
  8: {                         //cnf
  4: {                         //osc
  4: 10,                       //alg: AES-CCM-16-64-128
  0: "<osc-id>",               //id
  2: "ca5….4dd43e4c"           //ms
}}}
```

3.6.3.5 Device Handover

A device handover allows a new KNX IoT Client to get admin rights ("if.sec") to configure new access tokens. This method can be used if an already configured KNX IoT device needs to be integrated into another system.

The MaC or an existing KNX IoT Client with admin rights ("if.sec") can instruct a KNX IoT device to open a PASE enrollment window for a predefined time. The MaC SHALL generate a new temporary random password according to clause 3.6.6.6. The new KNX IoT Client gets the temporary password from the MaC and uses the password for a new PASE enrollment, according to 3.6.3.

User interfaces such as a local display or manufacturer-specific tools are other possible mechanisms to get a temporary password from a KNX IoT device to open a PASE enrollment window.

MaC configures a temporary password for PASE enrollment, valid for 60 seconds.

REQ:
POST coap://{ipv6-unicast}/.well-known/knx/spake
(Content-Format: application/cbor (60)), OSCORE(code(PUT), kid(<osc-id>)))

Payload:
{ 8: "ABCD1234", 9: 60 }

RES:
2.04 CHANGED

If the KNX IoT device does not support the configuration of new passwords, the device SHALL return a "4.00 Bad Request" response code.

If a member is a market as OPTIONAL, then a client can omit the member in the request, However, the server SHALL support the member handling.

Table 54 – SPAKE2+ Handover Password JSON keys and the CBOR mapping

JSON Key	CBOR Key	CBOR Type	Support	Description
"pw"	8	byte string	**M**	The temporary password for PAKE key enrollment. In the case the "pw" value is an empty string "", the KNX IoT device SHALL reuse the default PASE password,
"time"	9	unsigned	O	Password validity time in seconds. Default timeout, if omitted in the request, SHALL be 120 sec. Maximum timeout: 60 min.

3.6.4 Message Replay Protection

A KNX IoT device receiving an OSCORE request SHALL implement replay protection and validate message freshness according to [RFC8613] clause 7.4.

The general rule is that the first time a server receives an OSCORE unicast GET, DELETE, PUT, or POST request from an unsynchronized client, the server SHALL respond with an OSCORE protected CoAP response code "4.01 Unauthorized" with an Echo option according to [RFC9175] clause 2.3.

This is due to the replay window not yet being set up by the receiving KNX IoT device. Upon receiving a "4.01 Unauthorized" response code with the Echo option, the client SHOULD resend the original request by adding the Echo option with the received Echo option value.

The server SHALL NOT check the sequence number if an Echo option is present. This allows a client to update its security context with the server if the server has lost the sequence number. However, the client can use the Echo option value from the server only once with an invalid sequence number. Otherwise, the server responds with a new Echo option.

KNX IoT devices MAY trust the first authenticated multicast message from an unsynchronized peer by default, which is configurable (see clause 3.6.4.1.3).

3.6.4.1.1 Sender Security Context

A receiving KNX IoT device SHALL store received OSCORE message sequence numbers from each publishing sender security context, so that replay protection is enabled after receiving the first message. This approach has lower latency for less security-sensitive applications such as lighting. However, this method is susceptible to accepting a replayed message after a device reboot (see clause 3.6.4.1.3).

The OSCORE message sequence number and replay window SHALL be maintained with the configured security context (configured access token = pre-shared key), the OSCORE sender "kid", and the "kid_context".

The sending KNX IoT device (Publisher or client) SHALL use the OSCORE input material identifier (osc-id) and conextId from the access token for the OSCORE "kid" and the "kid_context" for unicast and multicast requests. The "kid_context" is used on the receiving device to distinguish security contexts. Therefore, the "kid_context" SHALL differ among KNX IoT devices with the same OSCORE credentials.

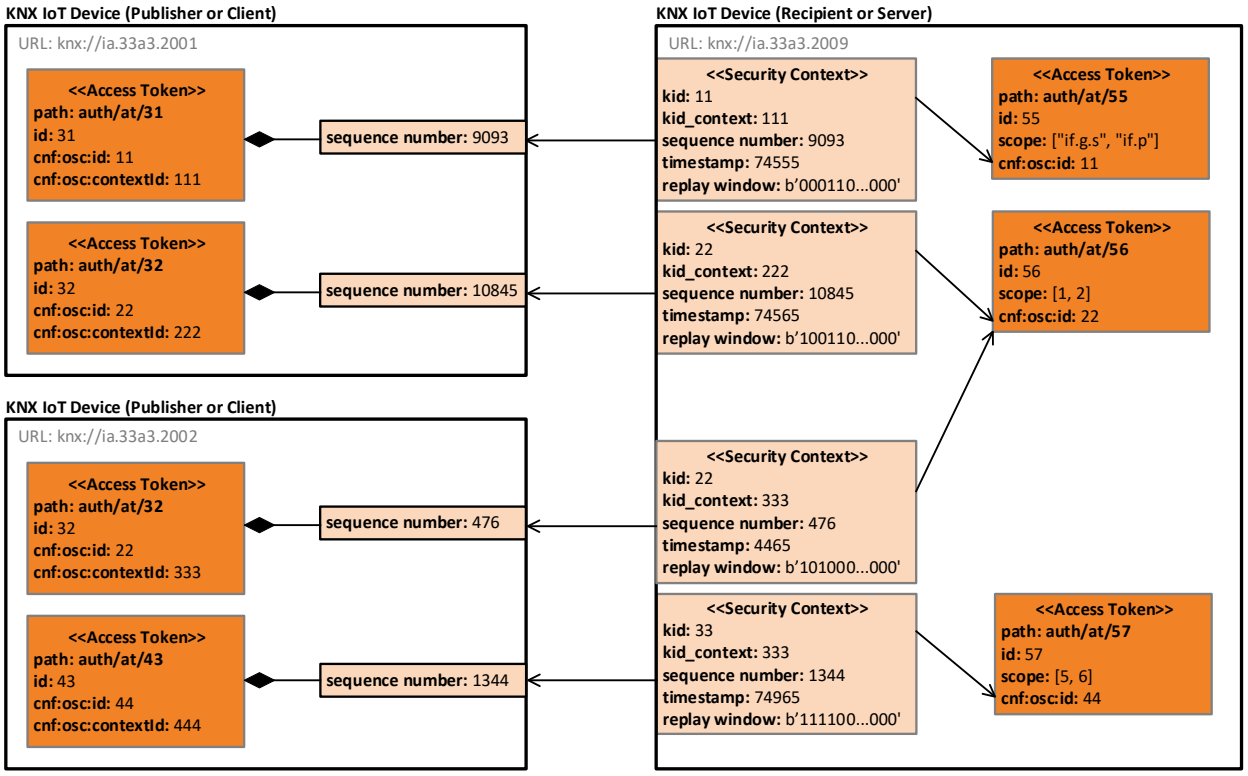


Figure 25 – Security Context

3.6.4.1.2 Message Sequence Number

The receiving KNX IoT device (Recipient or server) SHALL persist per publishing sender security context the received OSCORE message sequence number into non-volatile memory. If this is not possible, the device MAY synchronize the OSCORE message sequence number from configured peers (e.g., subscriptions or Function Point Publisher Table) based on a challenge-response mechanism.

A KNX IoT device SHALL support a list for S-Mode security context sequence number counters and, in addition, an independent list for unicast read/write/subscription sequence number counters.

The MaC SHALL NOT configure more sender security contexts than the maximal supported number of S-Mode sequence number counters of the sending KNX IoT device.

It is RECOMMENDED to support a sequence number counter for each subscription.

The KNX IoT device that sends an OSCORE message (unicast or multicast) SHALL ensure that the sequence counter does not roll back and SHALL persist that the sequence number counter in non-volatile memory for each security context. This means the sequence counter SHALL be strictly monotonic increasing for a given sender security context after the device restart.

If the sender sequence number exceeds the maximum configured value, the endpoint SHOULD continue with 0. This definition contrasts with [RFC8613] clause 7.2.1. However, if a KDC is present in the system, it SHALL configure a new security context before this happens. The maximum sequence counter SHALL be 0xFFFF FFFF = 2^{32}-1 (1 message per second = ~138 years).

A KNX IoT device SHALL generate a response to an OSCORE unicast request according to [RFC8613] clause 8.3 and reuse the sequence number from the sender. This reduces the consumption of sequence numbers on the responding device.

The receiving device SHALL check the OSCORE message sequence counter with the configured replay window (min = last received message + 1, max = last received message + configuration in auth/o/replwdo).

3.6.4.1.3 Sequence Number Synchronization

The sequence number synchronization mechanism MAY be used in combination with multicast runtime communication to update the security context sequence number.

When receiving a multicast request from a Publisher for the first time or the receiving device has lost synchronization, e.g., after reboot, the receiving device is not synchronized with the Publisher's sender sequence number and cannot verify if a request is fresh.

More security-sensitive applications MAY want to synchronize the sequence number before processing the OSCORE message. However, this is an application-specific configuration that can generate high communication traffic and drain battery capacity (e.g., light switch) and, therefore, SHALL be configurable with the "sns" metadata value (see clause 2.5.11.3, "Metadata Query Parameter "m").

If the application requires message freshness, then the receiving device SHALL (re-)synchronize the Publisher's sender sequence number based on the CoAP Echo approach described in [RFC9175] clause 2.3. A Publisher KNX IoT device SHALL support a challenge-response based on the Echo option if a peer requests it.

A KNX IoT device SHALL use the same credentials for the challenge-response request that initially triggered the (re-)synchronization. KNX IoT devices SHALL accept OSCORE message sequence number synchronization requests from all peers that use credentials (sender context) configured in the access control list (see clause 3.5.4 "Device Access Control List").

Sequence number synchronization.

REQ:
POST coap://{ipv6-multicast}/.knx
(Content-Format: application/cbor (60), OSCORE(code(POST), kid(<osc-id>),kid_context(<contextID>), sequence-number(10)))
Payload:
{ 4: <*KNX Individual Address*>, 5: { 6: <st>, 7: <ga>, 1: <value> } }

RES:
4.01 UNAUTHORIZED (Echo: 1234)

REQ:
POST coap://{ipv6-unicast}/.knx
(Content-Format: application/cbor (60), Echo(1234), OSCORE(code(POST), kid(<osc-id>), kid_context(<contextID>), sequence-number(10)))
Payload:
{ 4: <*KNX Individual Address*>, 5: { 6: <st>, 7: <ga>, 1: <value> } }

The following sequence diagram shows a sequence number synchronization example after receiving a multicast OSCORE message.

Upon receiving a request from a particular Publisher for the first time, the receiving device processes the message. Still, it does not deliver it to the application, even if it is valid. Instead, the receiving device replies with a random delay to the Publisher with an OSCORE protected "4.01 Unauthorized" response code, including only the Echo option value. This response SHALL be delayed (jitter) to avoid network congestion (see Table 55, "auth/o/osndelay").

Since this response is protected with the security context used in the previous request, the Publisher will consider the response valid upon successfully decrypting and verifying it.

Suppose the Publisher receives a "4.01 Unauthorized" response code that includes an Echo option and originates from a verified peer. In that case, it SHALL re-send an unconfirmed unicast echoing request to the same peer. The echoing message SHALL contain the sequence number previously used in the latest multicast message and the Echo option value from the response. The Publisher SHALL NOT send the request, including the Echo option, over multicast.

If other devices also want to (re-)synchronize with the same Publisher. The publishing device SHALL reuse the same sequence number from the latest multicast notification request in combination with the Echo option value from the corresponding device.

The Publisher maybe sends further multicast messages before echoing the "4.01 Unauthorized" response code from the receiving KNX IoT device, for example, since the response was delayed (jitter) on the receiving device. The Publisher SHALL take the latest (highest) sequence number used in the last multicast message for the unicast echoing message. The Recipient KNX IoT device SHALL discard duplicate messages before processing the message in the application.

If the KNX IoT device does not reply, the Publisher SHOULD wait for at least 2000 milliseconds + sleep cycle (non-sleepy devices = 0, sleepy devices = see clause 2.6.1.2.4.1) before the next retry.

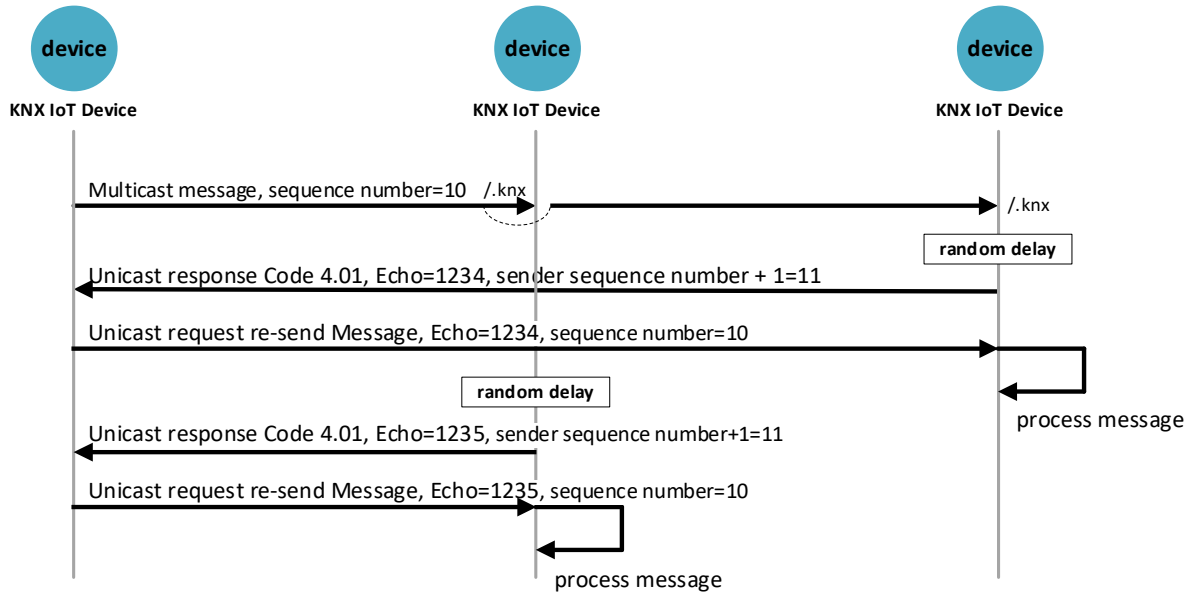


Figure 26 – OSCORE Sequence Number Synchronization

Table 55 specifies the MANDATORY ("M") resources for sequence number synchronization on a KNX IoT device.

Table 55 – Resources for sequence number synchronization

Resource path	rt & Data Types	Format	Method	Support	Request/Response	Notes
auth/o/replwdo	:dpt.value2uCount	cbor	GET PUT	**M**	Content-Format: application/cbor Payload: { 1: 32 }	The replay window is the maximum number of previously processed messages to accept from a given device and credential.

Resource path	rt & Data Types	Format	Method	Support	Request/Response	Notes
auth/o/osndelay	:dpt.timePeriodMsec	cbor	GET PUT	**M**	Content-Format: application/cbor Payload: { 1: 1000 }	Maximum delay (jitter) in milliseconds before sending a read message counter request (e.g., after receiving a multicast message). The default value is 1000 ms.

3.6.5 Message Processing

This clause describes the processing of OSCORE request and response messages in the context of KNX IoT according to [RFC8613].

If a KNX IoT device wants to send a request message, the client SHALL perform the following steps to create an OSCORE request:

- The sending KNX IoT device (client) queries the Function Point Table and SHALL use the configured access token for message encryption. The access token configuration for the OSCORE input material identifier (osc-id) SHALL be used for the OSCORE "kid", and the "contextID" SHALL be used for OSCORE "kid_context".
- Process steps according to [RFC8613] clause 8.1.

The receiving KNX IoT device (server) SHALL perform the following steps to validate the incoming OSCORE request:

- The KNX IoT device queries the access control list to find the corresponding access token item. The following criteria SHALL be applied to find the corresponding access token:
    - S-Mode unicast/multicast request: The access token OSCORE input material identifier (osc-id) SHALL correspond with the OSCORE "kid" in the request. The scope claim SHALL contain a *Group Address*.
    - Point read/write/subscribe unicast request: The access token OSCORE input material identifier (osc-id) SHALL correspond with the OSCORE "kid" in the request. In addition, the scope claim SHALL contain an interface type configuration (see clause 2.5.3).
- Process steps according to [RFC8613] clause 8.2.
- The KNX IoT device SHALL validate permissions before accessing Points (see clause 3.5.3).

In the case the KNX IoT device (server) sends a response message, the client SHALL perform the following steps to create an OSCORE response:

- Process steps according to [RFC8613] clause 8.3. The KNX IoT device SHALL use the AEAD nonce from the request and SHALL NOT include Partial IV in the message.
- Unicast response: Use the access token (credentials) from the request. The KNX IoT device SHALL use an empty byte string for the OSCORE sender ID (kid = h'').

The receiving KNX IoT device (client) SHALL perform the following steps to validate the incoming OSCORE response:

- Unicast response: Access token SHALL contain the corresponding credentials that match the previous request message.
- Process steps according to [RFC8613] clause 8.4.

3.6.6 OSCORE Cipher Suites

3.6.6.1 Key Derivation Function

A Key derivation function derives several secrets from one secret, the so-called Master Secret. KNX IoT devices SHALL support HKDF with SHA-256 according to clause 4.1 option 2 [NIST 800-56C] as a simple deterministic key derivation function based on the HMAC message authentication code.

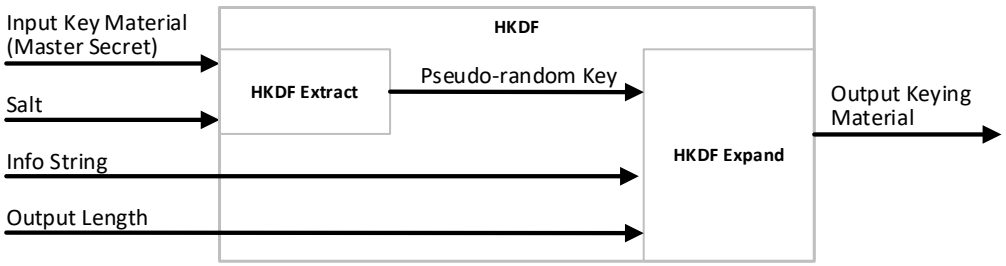


Figure 27 – HKDF

HKDF denotes the composition of the expand and extract functions as defined in [RFC5869], and the Master Secret is used as Input Key Material. The Master Secret ("ms" in table 3.6.2 "OSCORE Key Configuration Resource Object") SHALL be a random value of at least 16 bytes and, at most, 32 bytes.

The OSCORE security context parameters Sender Key, Recipient Key, and Common IV SHALL be derived from the HKDF input parameter. The Info String provides information to identify the security context. The Info String SHALL be a CBOR array according to [RFC8613] clause 3.2.1.

3.6.6.2 Password-based Key Derivation Function (PBKDF)

In practice, there is no significant difference between HKDF and PBKDF. HKDF has been explicitly designed for non-password-based key derivation, whereas using PBKDF2 with an iteration count is formally outside of the original HKDF parameters.

PBKDF2 [RFC2898] is a simple cryptographic key derivation function resistant to dictionary attacks and rainbow table attacks. PBKDF2 applies a pseudorandom function, such as a hash-based message authentication code (HMAC), to the input password along with a salt value and repeats the process many times to produce a derived key, which can then be used as a cryptographic key in subsequent operations.

The following list defines the MANDATORY cipher suites for the PASE security context of KNX IoT devices:

- PBKDF2 random value size: 32 bytes (secure random number)
- Salt: Device-specific random value of at least 16 bytes and at most 32 bytes.
- Count (PBKDF iterations): min=1000 and max=100000
- Total output length for PASE PBKDF2 (dkLen) = 40 + 40 bytes (w0s | | w1s) = 80 bytes.

3.6.6.3 OSCORE Security Context

KNX IoT devices that exchange messages need a shared security context for encryption and decryption on both sides. The OSCORE security context [RFC8613] defines a set of parameters necessary to perform cryptographic operations. The following list defines the MANDATORY cipher suites for the OSCORE security context of KNX IoT devices:

- AEAD Algorithm: AES-CCM-16-64-128
- Key Derivation Function: HKDF SHA-256 (see clause 3.6.6.1)

3.6.6.4 SPAKE2+ Enrollment Configurations

This clause specifies SPAKE2+ configuration parameters for password-authenticated OSCORE access token enrollment. All KNX IoT devices supporting OSCORE access token enrollment SHALL support the following cipher suite according to SPAKE2+ [13]: SPAKE2+_P256_SHA256_HKDF_HMAC.

It is assumed that the Initiator and the Responder know all relevant parameters (w0, key length L etc.) corresponding to the Password. The following list defines additional SPAKE2+ parameters such as elliptic curve points (M and N) etc.:

- SPAKE M Constant SPAKE2+ [13]:
  0x04886e2f97ace46e55ba9dd7242579f2993b64e16ef3dcab95afd497333d8fa12f5ff355163e43ce224e0b0e65ff02ac8e5c7be09419c785e0ca547d55a12e2d20
- Len(M): 65
- SPAKE N Constant SPAKE2+ [13]:
  0x04d8bbd6c639c62937b04d997f38c3770719c629d7014d49a24b4f98baa1292b4907d60aa6bfade45008a636337f5168c64d9bd36034808cd564490b1e656edbe7
- Len(N): 65
- Context: knxpase
- idProver SPAKE2+ [13]: Len(idProver) = 0x0000000000000000
- idVerifier SPAKE2+ [13]: Len(idVerifier) = 0x0000000000000000

The method for computing w0s and w1s SHALL be as follows:

w0s || w1s = PBKDF(len(password) || password || len(idProver) || len(idVerifier))

3.6.6.5 Transcript Computation (TT)

The protocol transcript (TT) is known by the Initiator and the Responder. Both parties use the TT to derive the shared symmetric key according to SPAKE2+ [13] clause 3.3.

The idProvider and the idVerifier are unknown during the PASE enrollment. Therefore, an empty octet string represented as an eight-byte little-endian number (0x0000000000000000) for idProver and idVerifier SHALL be used to indicate that no identities are present.

After exchanging PASE parameters, the Initiator and the Responder can compute Z and V, which are known values on both sides.

The following example depicts the protocol transcript encoding for the PASE enrollment.
TT = len(Context) || knxpase || 0x0000000000000000 || 0x0000000000000000 || len(M) || 0x04886e2f97ace46e55ba9dd7242579f2993b64e16ef3dcab95afd497333d8fa12f5ff355163e43ce224e0 b0e65ff02ac8e5c7be09419c785e0ca547d55a12e2d20 || len(N) || 0x04d8bbd6c639c62937b04d997f38c3770719c629d7014d49a24b4f98baa1292b4907d60aa6bfade4500 8a636337f5168c64d9bd36034808cd564490b1e656edbe7 || len(shareP) || shareP || len(shareV) || shareV || len(Z) || Z || len(V) || V || len(w0) || w0

3.6.6.6 Password

This clause defines the password for the OSCORE token configuration procedure using a shared password as proof of possession for KNX IoT devices in the default configuration state. The password character definition serves the following purposes:

- Minimize typing (or printing) errors since the Initiator (e.g., MaC) maybe allow users to enter a password manually.
- Alphanumeric encoding of passwords in QR codes.

The following characters from an alphabet of 32 numbers and upper-case letters SHALL be used: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, G, H, J, K, L, M, N, P, R, S, T, U, V, W, X, Y]. Hence, the following letters SHALL NOT be used for passwords: [I, O, Q, Z].

The KNX IoT device password SHALL be composed of at least six characters but at most 32 on both sides characters. To defend against brute-force attacks, the KNX IoT device SHALL limit authentication retries after a predefined number of consecutive invalid access attempts, e.g., wrong password. The KNX IoT device SHALL block further attempts. Authentication attempts SHALL be limited to at most one attempt per 5 seconds on average. This requires over 2,5 years of effort for an attacker to have a 50 % chance of success with a 6-character password.

The password SHOULD only be accessible with some form of physical proximity as proof of possession and SHALL NOT be accessible from a network interface (e.g., IP or CoAP). The password MAY be printed on a device (e.g., static QR code) or MAY be conveyed via NFC. Passwords, for example, in QR codes, SHOULD be hidden when installed. The password SHOULD be generated dynamically, for example, in dynamic QR codes or NFC messages. However, NFC readout of a static password SHALL NOT be possible if the commissioning mode is inactive, for example, when the device is in the packaging.

4 Software Update

4.1 Introduction

Software (firmware) updates are essential to keep installations healthy. The demand for shortening time to market for new applications and the accelerated pace of network and security standard evolution mandates the need for more frequent updates.

The KNX IoT device software update functionality is designed to meet the specific needs of devices with limited computing power and memory. Software updates require power, so special consideration must be given to power management. In addition, data throughput will determine the time it takes for updates and the energy used to execute them.

Therefore, image size is a key factor for a software update. KNX IoT supports an image-based software update strategy. This is achieved by encoding only the differences between a reference and a target image. For example, a package MAY contain only the communication stack that needs to be updated. However, the application software remains on the current version. This also enables the distribution of a software package to many of the same KNX IoT devices in an installation, for example, via multicast messages.

After the software package download, the KNX IoT device notifies the Software Update Server that the download is complete and it is ready to apply the downloaded software package ("swu/state" = downloaded).

In the last step, the Software Update Manager (e.g., MaC or Software Update Server) activates software download and updates the software.

The Software Update Manager (initiator) decides which software update mode to apply and either provides the Software Update Client with an URL to perform the download from a Software Update Server or directly sends the software to the KNX IoT device. Hence, the software update process looks as follows:

1. Software Update Manager activates software download.
2. Software Update Client (KNX IoT device) downloads the software package from the Software Update Server (PULL), or the Software Update Client writes the software package to the KNX IoT device (PUSH).
3. The KNX IoT device reports when finished.
4. The Software Update Manager decides what to do next and, when needed, sends a request to the Software Update Client to perform the update.
5. Software Update Client attempts to apply the software package and reports the status.

4.2 Software Update Client Resource (swu)

The Software Update Manager operates software updates on the Software Update Client *Functional Block* (SWU) which SHALL be implemented on a KNX IoT device.

A Software Update Manager MAY configure the SWU to fetch (PULL) a software package from a dedicated server instead of pushing (PUSH) software packages to the KNX IoT device.

If download speed is essential and the KNX IoT device supports additional protocols besides CoAP/UDP, then the server MAY choose one of those protocols (e.g., CoAP(s)/TCP) for software downloads. However, the Software Update Server SHALL support CoAP and block-wise transfer [RFC7959] for software downloads. Manufacturers SHOULD consider further aspects for secure software updates. Examples are how to manage a software update repository at the server side (which may include user interface considerations), the techniques to provide additional application layer security protection of the software package, how many versions of software images to store on the KNX IoT device, and how to execute the software update process considering the hardware specific details of a given KNX IoT product. These aspects are out-of-scope of this document.

However, a KNX IoT device SHALL verify the authenticity and integrity of a software package before applying a software package. Consequently, the software package SHALL contain package size, authenticity, and integrity information, for example, in a manufacturer-specific header section. The package size is used as an indication of when to start the software validation step.

A device usually has memory for two software packages (active and non-active): the currently active software and the newly uploaded software package or the previous (old) software package after the newly uploaded package has been activated. The new software download overwrites the non-active package (see swu/pkgname).

Table 56 specifies the MANDATORY ("M") or OPTIONAL ("O") software update resources (swu) and the corresponding resource path names that SHALL be used. (PUT) is OPTIONAL since the property value is preconfigured or only one option is supported.

Table 56 – Resources for software update

Resource path	rt & Data Types	Format	Method	Sup-port	Request/Response	Notes
swu	:fb.swu	link-format	GET	**M**	Content-Format: application/link-format Payload: </a/swu>;ct=60, </swu/pkgname>;ct=60, </swu/pkgbytes>;ct=60, </swu/pkgv>;ct=60, </swu/update>;ct=60, </swu/state>;ct=60, </swu/result>;ct=60, </swu/lastupdate>;ct=60, </swu/method>;ct=60	Response contains a list of points that belongs to the swu *Functional Block*.
a/swu	:dpt.file	octet-stream	PUT	**M**	Content-Format: "application/octet-stream" Payload: Manufacturer-specific binary file	Manufacturer-specific software package. It overwrites the oldest package version that exists on the device.
swu/pkgname	:dpt.varString8859_1	cbor	GET	**M**	Content-Format: application/cbor Payload: { 1: packagename }	The package name of the non-active software package. Returns response code "4.04 Not Found" if no non-active package is available (e.g., factory new device).

Resource path	rt & Data Types	Format	Method	Sup-port	Request/Response	Notes
swu/pkgqurl	:dpt.url	cbor	GET (PUT)	O	Content-Format: application/cbor Payload: { 1: swus.knx.local/firmware }	URL where to send a software update query request periodically. The URL format is defined in RFC 3986. The initial value of the mrt metadata value is 24 h (86 400 s)
swu/pkgbytes	:dpt.value 4uCount	cbor	GET	**M**	Content-Format: application/cbor Payload: { 1: 96325 }	Indicates already saved bytes in the non-volatile memory of the software package. The value SHALL be >0 only while swu/state= Downloading.
a/swu	NA	cbor	POST	O	Content-Format: application/cbor Payload: see clause 4.3, "Software Update Modes".	Triggers a software update query request (PULL on Software Update Server).
swu/pkgv	dpt.versio n	cbor	GET	**M**	Content-Format: application/cbor Payload: { 1: [1,2,3] }	The version of the newly available software package if the State resource is in state downloaded. Otherwise returns the response code "4.04 Not Found".

Resource path	rt & Data Types	Format	Method	Sup-port	Request/Response	Notes
swu/update	:dpt.timePeriodSecZ	cbor	PUT	**M**	Content-Format: application/cbor Payload: { 1: 0 }	Trigger for upgrading the firmware. The value contains a defer period in seconds (0=update immediately, max="swu/maxdefer"). The resource is executable only when the swu/state resource is in the downloaded state (swu/pkgname contains a filename). GET = remaining time or Z8 status = OSV. If the download is ongoing, the device SHALL return a response code "4.00 Bad Request".
swu/state	:dpt.dldState	cbor	GET	**M**	Content-Format: application/cbor Payload: { 1: 0 }	Indicates the current state concerning this software update. This value is set by the SWU. 0: Idle (before downloading or after successful updating) 1: Downloading (the data sequence is on the way) 2: Downloaded 3: If the "swu/update" is executed (true), the "swu/state" changes from Downloaded to Upgrading. If the update is successful, the "swu/state" changes to Idle. If the update fails, the status changes to Downloaded.

Resource path	rt & Data Types	Format	Method	Sup-port	Request/Response	Notes
swu/result	:dpt.updateResult	cbor	GET	**M**	Content-Format: application/cbor Payload: { 1: 0 }	Contains the result of the software update or download. 0: Initial value. Once the updating process is initiated (Download /Update), this Resource SHALL be reset to the initial value. 1: Software updated successfully. 2: Not enough flash memory for the new software package. 3: Out of RAM during downloading process. 4: Connection lost during downloading process. 5: Integrity check failure for the newly downloaded package. 6: Unsupported package type. 7: Invalid URL. 8: Software update failed. 9: Unsupported protocol.
swu/lastupdate	:dpt.varString8859_1	cbor	GET	**M**	Content-Format: application/cbor Payload: { 1: 2020-04-12T2 3:2 0:50.52Z }	Time of the last software update according to IETF RFC 3339. Initial value: date of manufacturing.

Resource path	rt & Data Types	Format	Method	Sup-port	Request/Response	Notes
swu/method	:dpt.transferMethod	cbor	GET (PUT)	**M**	Content-Format: application/cbor Payload: { 1: 2 }	The KNX IoT device (SWU) uses this resource to indicate its support for transferring software images to the client either via the Package Resource (=push) or via the Package URI Resource (=pull) mechanism. 0: Pull only 1: Push only 2: Both. In this case, the Update Server MAY choose the preferred mechanism for conveying the software package to the KNX IoT Client.
swu/maxdefer	:dpt.timePeriodSec	cbor	GET PUT	O	Content-Format: application/cbor Payload: { 1: 0 }	Maximum Defer Period The maximum number of seconds a software update can be deferred. When this period is over, the device will automatically update a new package. If the value is 0, an automatic update is not allowed.

Resource path	rt & Data Types	Format	Method	Sup-port	Request/Response	Notes
swu/protocol	:dpt.protocols	cbor	GET (PUT)	**M**	Content-Format: application/cbor Payload: { 1: 0 }	Protocol configuration. The following list defines the protocol enumeration: 0: Unicast CoAP with the additional support for OSCORE (as defined in RFC 7252) and block-wise transfer. CoAP is the default setting. 1: CoAPS (as defined in RFC 7252) with optional support for block-wise transfer 2: CoAP with OSCORE over TCP (as defined in RFC 8323) 3: CoAP over TLS (as defined in RFC 8323) 254: Manufacturer specific

4.3 Software Update Modes

4.3.1 Overview

KNX IoT defines the following two software download modes:

- PUSH: The Software Update Server initiates the download and performs write operations on the a/swu resource. This usually means that the block-wise transfer prevents other CoAP requests on the KNX IoT device until the write operation completes.
- PULL: the Software Update Server indicates from where the client should download the software package by configuring the "swu/pkgurl" (package URI) resource on the KNX IoT device. The KNX IoT device (client) then performs the download asynchronously while still being able to handle CoAP requests.

Although the Software Update Manager selects the download method, it is RECOMMENDED to use PULL transfers exclusively during normal operation of an installation.

Depending on the configured protocol, the corresponding access token configuration in the access control list SHALL be checked.

4.3.2 Software Update Query Resource Object

The following table defines the CBOR mapping for the different messages used for software updates between Software Update Manager, Software Update Server, and the SWU *Functional Block* (KNX IoT device). The attributes for these requests provide sufficient information to allow the KNX IoT device and the Software Update Server to determine the availability of a new software package.

A KNX IoT device firmware MAY be divided into several software packages. The KNX IoT device SHOULD use the software package type ("pkgt") to distinguish between firmware parts (e.g., "communication stack package" or "application package" etc.) and, therefore, the software package SHOULD contain the software package type information, for example, in the package header. The definition of package types and software package structure (header, image segment) is manufacturer specific.

Table 57 specifies the MANDATORY ("M") or OPTIONAL ("O") attributes for a software update resource object. If an attribute is marked as OPTIONAL, a client can omit the attribute in the request. However, the server SHALL support the attribute handling.

Table 57 – Software Update Query Resource Object members

JSON Key	CBOR Key	CBOR Type	Support	Description
"pkgn"	1	map	-	Package notification object.
"pkgq"	29	map	-	Package query object.
"result"	3	unsigned	O	Package query result.
"mid"	4	unsigned	**M**	Manufacturer ID of the KNX IoT device (see "dev/manufid").
"hwt"	5	unsigned	**M**	Software package product ID
"fwv"	7	[unsigned, unsigned, unsigned]	**(M)**	The current software version of the KNX IoT device (see "dev/fwv") in software update query request and response.
"prot"	8	unsigned	O	Supported download protocols (see "swu/protocol").
"defer"	9	unsigned	O	Package update deferred period (see "swu/maxdefer")
"pkgv"	10	[unsigned, unsigned, unsigned]	**(M)**	New software package version (see "swu/pkgv") in the software update notification.
"pkgurl"	11	text string	O	Package URL (see "swu/pkgurl") of the update server. The URL MAY be preconfigured.
"pkgt"	12	unsigned	O	Package type (manufacturer-specific number)
"pkgs"	13	unsigned	O	Package Size (kByte)
"po"	14	unsigned	O	Page offset (see clause 4.3.3)
"ps"	15	unsigned	O	Page size (see clause 4.3.3)

4.3.3 Software Update Query Parameter "p", "ps", and "pkg"

Software update query parameters provide the information about which part from a software update package is written (PUSH mode) to the KNX IoT device or is requested (PULL mode) from the Software Update Server.

In PUSH mode, the first PUT request to "a/swu" SHALL contain the query parameter "pkgs={bytes}" indicating the total size of the software update package. In subsequent requests, the KNX IoT device SHALL ignore the "pkgs=" query parameter.

In PULL mode, the client indicates with the query parameter "ps={byte size}" in the GET request the maximum file size (bytes) the device can handle in the response to the Software Update Server. If "ps" is missing, common values from CoAP option Block1 and Block2 are used.

The page offset "po" can be used in PULL mode (GET firmware) and in the PUT request to "a/swu" in PUSH mode. The query parameter "po" defines the number of bytes to skip. If "po" is omitted in the request, then the default offset SHALL be 0.

The following picture depicts a software update package that is split into two independent byte chunks. The first byte chunk starts at the beginning of the software update package (po=0). After receiving the first byte chunk, the software update continues with the next byte chunk and an offset "po" (po=<received bytes>). The page size "ps" is transferred to the device with CoAP block-wise transfer, and each "ps" byte chunk SHALL start with block number 0.

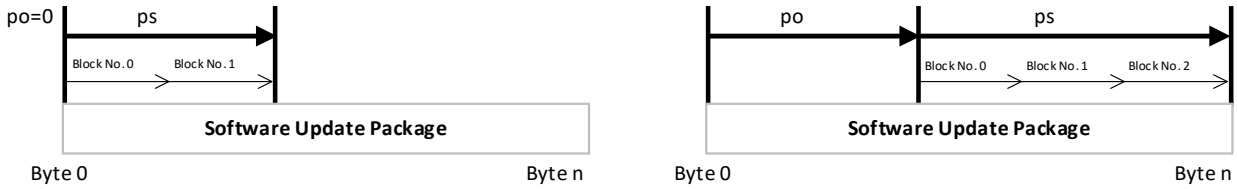


Figure 28 – Software Package Query Parameter

The following example shows a KNX IoT request that starts with an offset of 2056 bytes and indicates that it can handle a maximum of 1028 bytes.

Read software update package chunk from Software Update Server (incl. block-wise transfer).
REQ: GET coap://{ipv6-unicast}/firmware?po=2056&ps=1028 (Content-Format: application/octet-stream (42), block num 0 (block size: 64)) **Payload:** Uiow578oiqwhtq8745 **RES:** 2.04 CHANGED

4.3.3.1 Resume Software Update

Software update packages are often multiple kilobytes, sometimes exceeding one hundred kilobytes in size. If, for some reason, a software update is interrupted, like a disruption in connectivity or a device losing power, the software update needs a mechanism to resume interrupted transfers.

Suppose a KNX IoT device wants to continue reading a software package (PULL mode) from a Software Update Server, for example after a device restart, the device SHOULD request the next byte chunk with a byte offset "po" after the last successfully received byte ("swu/pkgbytes").

If a Software Update manager wants to resume an interrupted transfer, then it SHOULD read from the KNX IoT device the number of bytes already saved in nonvolatile memory ("swu/pkgbytes") and continue from there.

The Software Update Manager using PUSH mode SHALL indicate where to append byte chunks with "po" (po={"swu/pkgbytes"}) in the PUT request to "a/swu" or with a software notification request in PULL mode (see clause 4.3.4.2).

4.3.4 Software Update PULL

4.3.4.1 Basic requirements

KNX IoT devices MAY support Software Update PULL. If the PULL method is configured, then the KNX IoT device SHALL support Software Update Notifications if the Software Update Manager wants to trigger the software update process.

In addition, the SWU *Functional Block* MAY support checking the configured Software Update Server periodically with a Software Update Query for new software packages. The Software Update Server SHALL support block-wise transfer [RFC7959] since this is the mandatory software update transfer format for a KNX IoT device.

It is RECOMMENDED that sleepy KNX IoT devices make their best effort to optimize their wake-up intervals to optimize battery live time or other constrained power sources while retrieving a new software package. Therefore, KNX IoT devices SHOULD support block-wise transfer to recover and continue the download without having to start from the beginning of a given software package (see 4.3.5, "Software Update PUSH", e.g., "swu/pkgbytes"). This takes more time to complete the download process and SHALL be considered by the Software Update Manager, for example, if it wants to trigger the software update and restart all KNX IoT devices of a given installation simultaneously.

Since the SWU *Functional Block* MAY need to read the Software package in parts, it is RECOMMENDED that Software Update Servers maintain the cached package for at least 10 minutes after closure of the last package transfer so that the SWU *Functional Block* MAY come back to read different parts of a package file.

Software Update Clients SHALL abort retrying a transfer after three attempts and retry with a standard software update query request (e.g., "check available software" after 24h). The resulting error SHALL be logged in the "swu/result" Point.

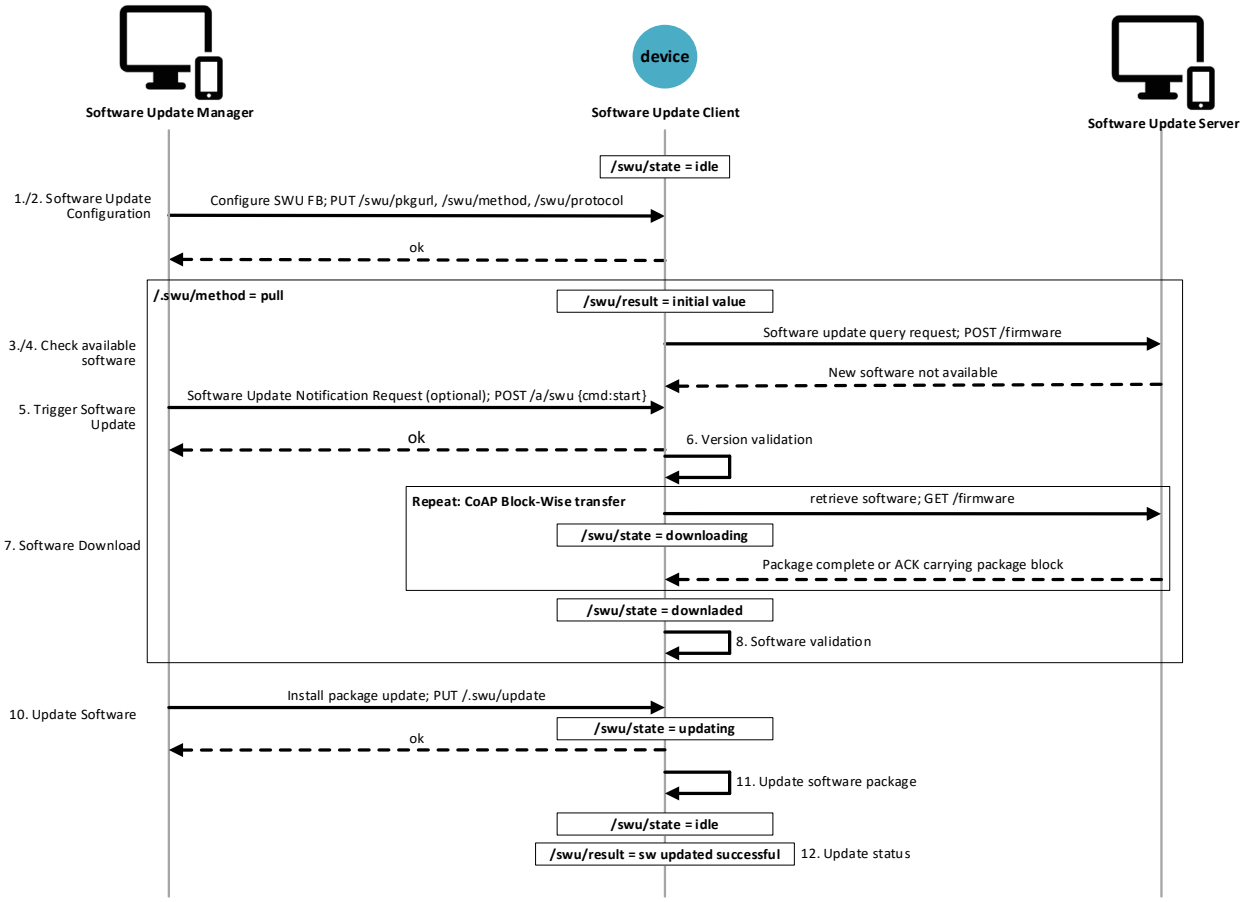


Figure 29 – Software Update PULL

The example depicted in the previous figure illustrates a successful message exchange between Software Update Manager/Server and a KNX IoT device. In this example, no messages are lost during transmission:

1. Software Update Manager configures the Software Update Server and PULL as an update method on the KNX IoT Client. This configuration can be done with a MaC over CoAP or out-of-band (e.g., NFC or BLE, etc.). In the case of CoAP, the Software Update Manager MAY have a configured device list or MAY discover KNX IoT devices, for example, with mDNS.
2. KNX IoT device determines Software Update Server to query ("swu/pkgurl").
3. KNX IoT device periodically checks for available software packages and queries the Software Update Server for a new version.
4. If Software Update Server has no new software available, then retry after the heartbeat period expires (see metadata). Else, continue with 6.
5. OPTIONAL: Software Update Manager triggers the KNX IoT device to start immediately with the software download.
6. The KNX IoT device validates whether to apply the new software version.
7. KNX IoT device downloads the software package with the configured protocol ("swu/protocol"), for example, with CoAP block-wise transfer (default) in chunks suitable to the maximum transmission size (MTU) or the adaption-layer fragmentation (e.g., 60-80 bytes for 6LoWPAN).
8. KNX IoT device validates the new software package (e.g., manufacturer signature, etc.)

9. OPTIONAL: KNX IoT device notifies (with subscription) the Software Update Server that the download is complete and it is ready to apply the downloaded software package ("swu/state" = downloaded).
10. Software Update Manager triggers the KNX IoT device to apply the software package.
11. KNX IoT device applies the software package.
12. KNX IoT device notifies (read or with Point Subscription) the Software Update Manager of having successfully applied the software package ("swu/result" = sw updated successfully).

4.3.4.2 Software Update Notification

Installations often have many of the same KNX IoT devices (same product and version). The Software Update Servers SHOULD avoid redownloading the same software package if it can determine that multiple KNX IoT devices are requesting or can be expected to request the same software package.

The software update notification command is used if a Software Update Manager wants to control or force the rollout of new software packages. The software update notification command SHALL contain the following attributes from Table 57:

- mid: software package manufacturer ID
- hwt: software package product ID
- pkgv: version of the new available package

The software update notification command MAY contain the following attribute:

- pkgt: software package type
- pkgurl: the URL from where to fetch new software package.
- pkgs: file size of the software package [kByte]
- defer: deferred software package installation after download.
- prot: Supported protocols for software download (default CoAP with OSCORE)
- po: Page offset (see clause 4.3.3)
- ps: Page size (see clause 4.3.3)

The following example does not contain a Package Update Deferred Period (defer) attribute; therefore, the package update SHALL NOT be installed automatically. In this case, the Software Update Manager wants to apply the new software package by sending a separate request to "swu/update" afterward. The following example uses the JSON Content-Format for readability:

Software Update Notification example (new software update available).

REQ:

```
POST coap://{ipv6-unicast}/a/swu
(Content-Format: application/json (50), Accept: application/json (50), OSCORE(code(PUT), kid(<osc-id>)))
```

Payload:

```
{
  "cmd": "start",
  "pkgn": {
  "mid": 123, "hwt": 0x456AB, "pkgt": 0, "pkgv": [1,2,3], "pkgs": 23, "prot": [6],
  "pkgurl": "coap://swus.knx.local/firmware:61630" }
}
```

RES:

```
2.04 CHANGED
```

4.3.4.3 Software Update Query

The KNX IoT device MAY check periodically for new available software packages by default. However, it SHALL be possible to disable automatic software updates and checks ("swu/pkgurl"–- metadata: heartbeat=0). If supported, then the SWU *Functional Block* SHOULD configure at least a minimal heartbeat of 24 hours and a maximum of three retries, for example, if the server does not reply, between two query requests per Software Update Server, unless a KNX IoT device loses its time, due to power loss or restart, etc. This reduces the burden on the Software Update Server, providing the service to many devices and supporting networking infrastructure.

Suppose the Software Update Server cannot immediately respond since it is busy; the server SHALL return response code "5.03 Service Unavailable" but uses the Max-Age option to indicate the number of seconds after which to retry.

The update notification query request SHALL contain the following attributes from Table 57:

- mid: software package manufacturer ID
- hwt: software package product ID
- hwv: device hardware version
- fwv: device software or firmware version.

The update notification query request MAY contain the following attributes:

- pkgt: software package type
- The update notification query response SHALL contain the following attribute:result: query result. The following example uses the JSON Content-Format for readability:

Software Update Query example (software update not available).

REQ:
POST coap://{ipv6-unicast}/firmware
(Content-Format: application/json (50), Accept: application/json (50), OSCORE(code(POST), kid(<osc-id>)))
Payload:
```
{
  "pkgq": {
  "mid": 123, "hwt": 0x456AB, "pkgt": 0, "hwv": [1,2,3], "fwv": [2,3,4] }
}
```

RES:
2.05 CONTENT (Content-Format: application/json (50))
Payload:
```
{ "result": 0 }
```

It may happen that during package query or package download, the SWU *Functional Block* encounters error conditions that eventually succeed through retrying the same operation more than once. Hence, the SWU *Functional Block* will repeat a Software Update Query request. Even if the Software Update Server can detect this, it SHALL NOT behave differently on any subsequent attempt compared to the first, unless a new software package becomes available in the meantime.

If the new software update is available, then the update query response SHALL contain the following attributes from Table 57:

- mid: software package manufacturer ID
- hwt: software package product ID

- pkgt: software package type
- pkgv: version of the newly available software package.
- pkgs: file size of the new software package [kByte]
- pkgurl: the URL from where to fetch the new software package.
- result: query result.

The update notification response MAY contain the following attribute:

- defer: deferred software package installation after download.

The following Software Update Query example defines a Package Update Deferred Period (defer) of 0 seconds in the response. Hence, the KNX IoT device SHALL automatically install the new software package after the download, except "swu/maxdefer" is set to 0. If defer is absent, the new software package SHALL NOT be installed automatically. The following example uses the JSON Content-Format for readability:

Software Update Query example (new software update available).

```
REQ:
POST coap://{ipv6-unicast}/firmware
(Content-Format: application/json (50), Accept: application/json (50), OSCORE(code(POST), kid(<osc-id>)))
Payload:
{
  "pkgq": {
  "mid": 123, "hwt": 0x0123AB, "pkgt": 0, "hwv": [1,2,3], "fwv": [2,3,4], "prot": 6 }
}

RES:
2.05 CONTENT (Content-Format: application/json (50))
Payload:
{
  "result": 1,
  "pkgn": {
  "mid": 123, "hwt": 0x0123AB, "pkgt": 0, "pkgv": [1,2,3], "pkgs": 23,
  "pkgurl": "coap://swus.knx.local/firmware:61630" }
}
```

4.3.5 Software Update PUSH

Software Update PUSH simplifies the KNX IoT device and puts the update logic to the Software Update Manager. Resource-constrained devices MAY NOT be able to implement Software Update PULL. In addition, with Software Update PUSH, the entire update process is managed by the Software Update Manager, including package downloads. The Software Update Manager can optimize the process, such as subsequent dynamic loads of packets based on a broader range of information. Therefore, a KNX IoT device SHALL support Software Update PUSH.

In the "PUSH" as well as in the "PULL" scenario, the KNX IoT device SHALL sum up the transmitted bytes until the last message that has been successfully received and SHOULD store this information into non-volatile memory ("swu/pkgbytes") before power-down. When a new data packet PUSH or PULL operation begins, the information SHOULD be checked, and the update process SHOULD continue after the last successfully stored message on the device.

The KNX IoT device SHALL change to fast-poll or preferably to always-on mode after receiving the first block of a software update package (PUT "a/swu") and make their best effort to optimize their wake-up intervals.

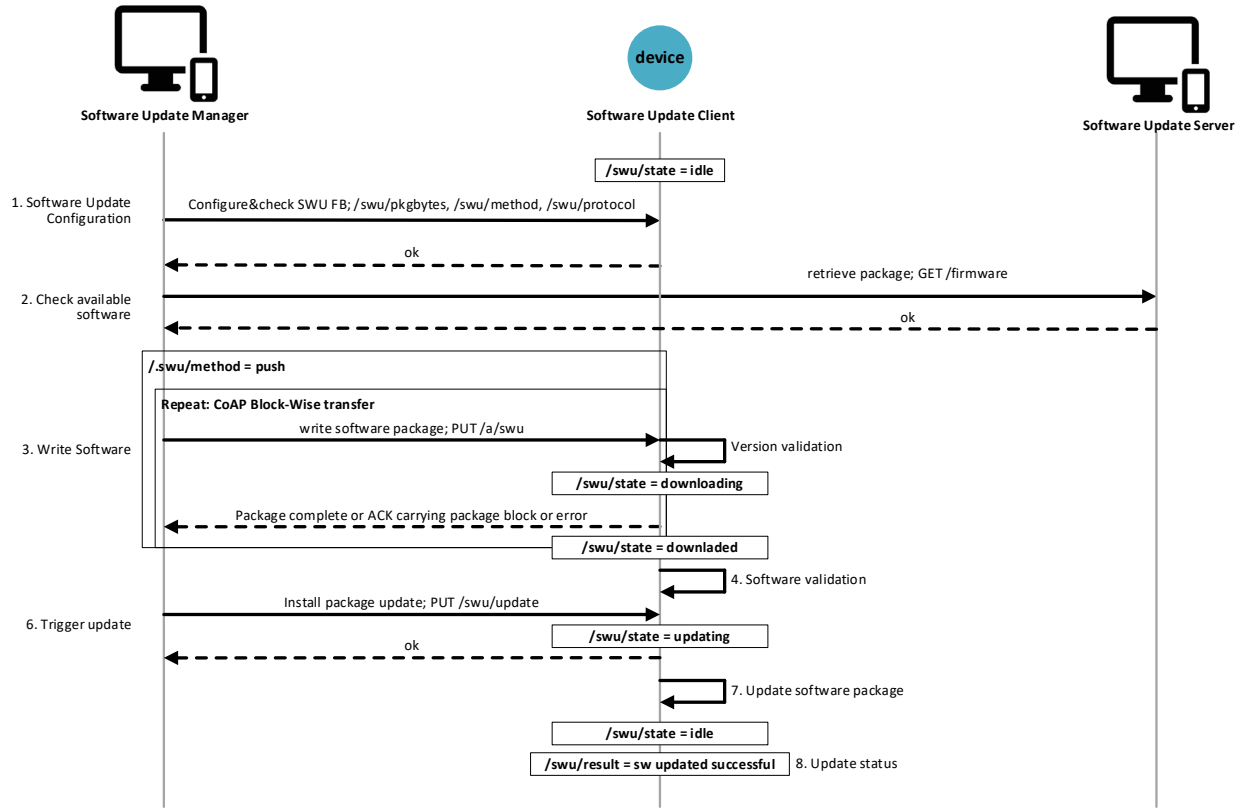


Figure 30 – Software Update PUSH

The example depicted in the previous figure illustrates a successful message exchange between Software Update Manager and a KNX IoT device. It is RECOMMENDED to use confirmed requests for all unicast communication. In this example, no messages are lost during transmission:

1. Software Update Manager configures PUSH as an update method on the KNX IoT Client. This configuration can be done with a MaC over CoAP or out-of-band (e.g., NFC, BLE, etc.). In the case of CoAP, the Software Update Manager MAY have a configured device list or MAY discover KNX IoT devices, for example, with mDNS.
2. Software Update Manager periodically checks for available software packages and queries the Software Update Server for a new version.
3. Software Update Manager writes a new software package to the KNX IoT device, for example, with CoAP block-wise transfer (default) in chunks suitable to the maximum transmission size (MTU) or the adaption-layer fragmentation (e.g., 60-80 bytes for 6LoWPAN).
4. KNX IoT device validates the new software package (e.g., manufacturer signature, etc.).
5. OPTIONAL: KNX IoT device notifies (with subscription) the Software Update Server that the download is complete and it is ready to apply the downloaded software package ("swu/state" = downloaded).
6. Software Update Manager triggers the KNX IoT device to apply the software package.
7. KNX IoT device applies the software package.
8. KNX IoT device notifies (client reads "swu/result" = sw updated successfully) the Software Update Manager of having successfully applied the software package.

5 Profiles

5.1 KNX IoT Point API Device

5.1.1 Default Configuration State

Table 58 defines features and resources for KNX IoT that a device SHALL ("M") or MAY ("O", for optional) support in the default configuration state according to [AN194]. A device in default configuration state MAY have a preset application configuration but SHALL have no installation-specific credential (security zone) configuration. Resources of the following device features SHALL be accessible without security credentials in the default configuration state. For example, a device in default configuration state SHALL reply to an "ep" ("/.well-known/core") device discovery.

Table 58 – Features in the default configuration state

Device Features	Support	Notes
Device discovery with DNS-SD and "ep"	M	2.6.1.2 "Device Discovery with DNS-SD" 2.6.1.3.6.1 "Endpoint Name "ep"" Affected resources: GET /.well-known/core
Read IDevID	M	3.3.1 "Manufacturer Device Certificates (IDevID)" Affected resources: GET auth/idevid
OSCORE configuration with TOFU (Trust On First Use)	M	3.6.3 "Password Authenticated Access Token Enrollment" Affected resources: POST /.well-known/knx/spake GET auth/at
LDevID simple enrollment with EST (PULL)	O	3.2.4 "Operational Device Certificate Enrollment (Pull Certificate)" Affected resources: a/sen

5.1.2 Commissioned Mode

Table 59 defines features for KNX IoT that a device SHALL ("M") or MAY ("O", for optional) support in normal operation.

Table 59 – Features for normal operation

Device Features	Support	Notes
Device and resource discovery	M	2.6.1 "Discovery" Affected resources: GET /.well-known/core

Device Features	Support	Notes
Application Layer Security with pre-shared keys (if.sec)	M	3.5.3 "Access Scope" 3.5.4 "Device Access Control List Resource (auth/at)" 3.6 "OSCORE Application Layer Security" Affected resources: GET, POST, DELETE auth/at/* GET auth/o (including OSCORE Properties in p/*)
Device individualization (if.sec, if.c)	M	2.5.5.5 Device Individualization Resource (.well-known/knx/ia) Affected resources: POST /.well-known/knx/ia
Read/Write Points (if.p, if.d)	M	2.5.11 "Parameter and Diagnostic Property Resource (p)" Affected resources: GET, PUT p/*
Pub/Sub with CoAP Observe (if.i, if.o)	M	2.5.11 "Parameter and Diagnostic Property Resource (p)" 2.5.12 "Subscription Resource (sub)" Affected resources: GET + Observe p/* DELETE sub/
Function Point Table configuration	M	2.5.7 "Function Point Table (fp)" Affected resources: GET, POST, DELETE fp/*/* GET, POST a/lsm Function Point Publisher Table MAY NOT be supported on specific device types such as switches (e.g., sleepy devices)
S-Mode Group Communication (if.g.s)	M	2.5.9 "S-Mode Messaging Resource (.knx)" Affected resources: POST .knx
Software update (PUSH)	M	4.3.5 "Software Update PUSH" Affected resources: GET, PUT swu/*
LDevID Enrollment (PUSH)	O	3.2.5 "Management Client as Registrar (Push Certificate)" Affected resources: auth/sen, or auth/skg
Transport Layer Security (if.sec)	O	3.2 "Device Identity Enrollment" 3.3 "Device Identity Certificates" 3.4 "Certificate " 3.5.3 "Access Scope" 3.5.2 "Trust List Resource (auth/crts)" 3.5.5 "Revocation List"

Device Features	Support	Notes
Software update (PULL)	O	4.3.4 "Software Update PULL" Affected resources: POST a/swu GET, PUT swu/*

3201

3202 ### 5.1.3 Device Resource List

3203 **Table 60 – Device Resource List**

Resource path	Entity type	Interface	Content-Format	Method	Support	Description
Discovery						
/.well-known/core	Collection		link-format	GET	M	List with device functional blocks
/.well-known/core?ep=	Item		link-format	GET	M	Device serial number
/.well-known/core?if=	Collection		link-format	GET	M	Filter: Interfaces
/.well-known/core?rt=	Collection		link-format	GET	M	Filter: Resource type
/.well-known/core?d=	Collection		link-format	GET	M	Filter: Group address
Device						
/.well-known/knx	Item		cbor, json	GET	M	API version & base path
/.well-known/knx	Action	if.sec,if.c	cbor	POST	M	Device reboot and reset
/.well-known/knx/ia	Action	if.sec,if.c	cbor, (json)	POST	M	Device individualization
/.well-known/knx/f	Item	if.c	cbor, (json)	GET	M	Device configuration fingerprint
/{base-path}/dev	Collection	if.ll	link-format	GET	M	List with device functional block Properties
/{base-path}/dev/{property-id}	Item	if.d	cbor, (json)	GET	M	Read Property value
/{base-path}/dev/{property-id}	Item	if.p	cbor, (json)	PUT	M	Write Property value
/{base-path}/dev/{property-id}?m=	Item	if.d	cbor, (json)	GET	M[a]	Filter: Read a selected list of metadata values
/{base-path}/dev/{property-id}	Item	if.p	cbor, (json)	PUT	M[a]	Write metadata value (with key)
/{base-path}/ap	Collection	if.ll	link-format	GET	M	List with application program functional block Properties
/{base-path}/ap/{property-id}	Item	if.d	cbor, (json)	GET	M	Read Property value
/{base-path}/ap/{property-id}	Item	if.p	cbor, (json)	PUT	M	Write Property value
/{base-path}/ap/{property-id}?m=	Item	if.d	cbor, (json)	GET	Ma	Filter: Read a selected list of metadata values
/{base-path}/swu	Collection	if.ll	link-format	GET	M	List with software update functional block Properties
/{base-path}/swu/{property-id}	Item	if.d	cbor, (json)	GET	M	Read Property value
/{base-path}/swu/{property-id}	Item	if.swu	cbor, (json)	PUT	M	Write Property value
/{base-path}/swu/{property-id}?m=	Item	if.d	cbor, (json)	GET	M[a]	Filter: Read a selected list of metadata values
/{base-path}/swu/{property-id}	Item	if.p	cbor, (json)	PUT	O[a]	Write metadata value (with key)
Actions						
/{base-path}/a/swu	Action	if.swu	octet-stream	POST	O	Trigger firmware update PULL
/{base-path}/a/lsm	Action	if.c	cbor	POST	M	Command Device Load State Machine
/{base-path}/a/lsm	Item	if.c	cbor	GET	M	Device Load State Machine status
/{base-path}/a/sen	Action	if.sec	cbor	POST	O	LDevID enrollment

Resource path	Entity type	Interface	Content-Format	Method	Support	Description
Security						
/.well-known/knx/spake	Item		cbor	POST	M	PASE
/.well-known/knx/spake	Item	if.sec	cbor	PUT	O	Password for device handover
/.well-known/knx/idevid	Item	if.d	pkcs7-mime	GET	O	Read the manufacturer's device certificate
/.well-known/knx/ldevid	Item	if.d	pkcs7-mime	GET	O	Write operational device certificate
/{base-path}/auth	Collection	if.ll	link-format	GET	O	List with/ security sub-resource
/{base-path}/auth/crts	Collection	if.sec,if.b	pkcs7-mime	POST	O	Write certificates
/{base-path}/auth/crts	Collection	if.ll	link-format	GET	O	Read certificate list (link to resources)
/{base-path}/auth/crts/{cert-id}	Item	if.sec	cbor	DELETE	O	Delete certificate
/{base-path}/auth/at	Collection	if.sec	cbor	POST	M	Write access tokens
/{base-path}/auth/at	Collection	if.ll	link-format	GET	M	Read access tokens list (link to resources)
/{base-path}/auth/at/{token-id}	Item	if.sec	cbor	DELETE	M	Delete access token
/{base-path}/auth/o	Collection	if.ll	link-format	GET	M	List with OSCORE functional block Properties
/{base-path}/auth/o/{property-id}	Item	if.p	cbor, (json)	GET	M	Read Property value
/{base-path}/auth/o/{property-id}	Item	if.sec	cbor, (json)	PUT	M	Write Property value
Messaging						
/.knx	Action	if.g.s	cbor	POST	M	KNX group communication
/.knx	Event	if.g.s	cbor	GET	M	KNX group communication (CoAP Observe)
Function Point Tables						
/{base-path}/fp/g	Collection	if.c,if.b	cbor	POST	M	Add or change group object items
/{base-path}/fp/g	Collection	if.ll	link-format	GET	M	Read group object list (link to resources)
/{base-path}/fp/g/{group-object-id}	Item	if.p	cbor	GET	M	Read group object item
/{base-path}/fp/g/{group-object-id}	Item	if.c	cbor	DELETE	M	Delete group object item
/{base-path}/fp/r	Collection	if.c,if.b	cbor	POST	M	Add or change recipient items
/{base-path}/fp/r	Collection	if.ll	link-format	GET	M	Read recipient list (link to resources)
/{base-path}/fp/r/{recipient-id}	Item	if.d	cbor	GET	M	Read recipient item
/{base-path}/fp/r/{recipient-id}	Item	if.c	cbor	DELETE	M	Delete recipient item
/{base-path}/fp/p	Collection	if.c,if.b	cbor	POST	O	Add or change publisher items
/{base-path}/fp/p	Collection	if.ll	link-format	GET	O	Read publisher list (link to resources)
/{base-path}/fp/p/{publisher-id}	Item	if.d	cbor	GET	O	Write publisher item
/{base-path}/fp/p/{publisher-id}	Item	if.c	cbor	DELETE	O	Delete publisher item
Functional blocks						
/{base-path}/f	Collection	if.ll	link-format	GET	M	Read functional block list (link to resources)
/{base-path}/f/{fb-id-instance}	Collection	if.ll	link-format	GET	M	Read functional block property list (link to resources)
Properties						
/{base-path}/p	Collection	if.c, if.b	cbor, (json)	POST	M	Add or change Property items
/{base-path}/p	Collection	if.ll	link-format	GET	0	Read Property list (link to resources)
/{base-path}/p/{property-path}	Item	if.d	cbor, (json)	GET	M	Read Property item
/{base-path}/p/{property-path}?lt=	Item	if.d	cbor, (json)	GET	M	Read Property item and observe with lifetime
/{base-path}/p/{property-path}	Item	if.p	cbor, (json)	PUT	M	Write Property item
/{base-path}/p/{property-path}?m=	Item	if.d	cbor, (json)	GET	M[a]	Filter: Read a selected list of metadata values
/{base-path}/p/{property-path}	Item	if.p	cbor, (json)	PUT	O[a]	Write metadata value (with key)
Subscriptions						
/{base-path}/sub	Collection	if.p	cbor, (json)	DELETE	M	Delete all device subscriptions, including CoAP observe
[a] Resource contains metadata (see clause 2.5.11.3).						

3204

5.2 CBOR Encoding

5.2.1 Function Point Tables, Functional Blocks, and Properties

Table 61 – Function Point, Functional Block, and Property CBOR encoding

cbor	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
json	id	value	cmd	status	sia	s	st	ga	cflag	
cbor	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**
json	url	href	ia	grpid	at					
cbor	**20**	**21**	**22**	**23**	24	25	26	27	28	29
json						fid	iid			pkgq
cbor	100 (d)	101 (e)	102 (f)	103 (g)	104 (h)	105 (i)	106 (j)	107 (k)	108 (l)	109 (m)
json			fault							
cbor	110 (n)	111 (o)	112 (p)	113 (q)	114 (r)	115 (s)	116 (t)	117 (u)	118 (v)	119 (w)
json		overridden								

5.2.2 Software Update Package Query

Table 62 – Software Update CBOR encoding

cbor	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
json		pkgn		result	mid	hwt	hwv	fwv	prot	defer
cbor	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**
json	pkgv	pkgurl	pkgt	pkgs	po	ps				
cbor	**20**	**21**	**22**	**23**	24	25	26	27	28	29
json										

5.2.3 Security

5.2.3.1 Access Token (RFC 8392)

Table 63 – Access Token CBOR encoding

cbor	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
json	id	iss	sub		exp	nbf	iat	cti	cnf	scope
cbor	**30**	**31**	**32**	**33**	**34**	**35**	**36**	**37**	**38**	**39**
json									profile	

5.2.3.2 Access Token Confirmation Methods (RFC 8747)

Table 64 – Access Token Confirmation Methods

cbor	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
json		jwk	jwe	kid	osc					

5.2.3.3 OSCORE Key Configuration incl. PASE

Table 65 – OSCORE Key Configuration

cbor	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
json	id	version	ms	hkdf	alg	salt	contextId		pw	time
cbor	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**
json	pa	pb	pbkdf2	cb	ca	rnd	it			

6 Examples

6.1 DEVICE POINT LIST EXAMPLES

6.1.1 Device Point List Example with OSCORE and (D)TLS

Resource Name	Interaction	rt & Data Types	if	Content-Format	Description
Common Services					
.well-known/core	ActionAffordance			link-format	Device and rt Discovery
.well-known/knx	ActionAffordance			cbor	Returns {prefix}
.well-known/knx/spake	ActionAffordance			cbor	PASE device bootstrapping
.well-known/knx/ia	ActionAffordance			cbor	Configure IA
.well-known/knx/f	PropertyAffordance			cbor	Returns a configuration fingerprint
Security					
auth			if.ll	link-format	
.well-known/knx/idevid	PropertyAffordance	:dpt.x509	if.d	pkcs7-mime	Manufacturer Device Certificate (X.509)
.well-known/knx/ldevid	PropertyAffordance	:dpt.x509	if.d	pkcs7-mime	Operational Device Certificate (X.509)
auth/crts	PropertyAffordance		if.ll	link-format	Read list with certificate IDs
auth/crts	PropertyAffordance		if.p if.sec	pkcs7-mime	Write CA Certificates
auth/at	PropertyAffordance		if.ll	link-format	Read list with access token IDs (e.g., OSCORE Credentials)
auth/at	PropertyAffordance		if.sec	cbor	Write access token IDs (e.g., OSCORE Credentials)
auth/at/{token-id}	PropertyAffordance		if.sec	cbor	Write access token (e.g., OSCORE Credentials)
Device Object					
dev		:fb.0	if.ll	link-format	
dev/sn	PropertyAffordance	:dpa.0.11 :dpt.serNum	if.d	json/cbor	Device serial number
dev/mid	PropertyAffordance	:dpa.0.12 :dpt.value2Ucount	if.d	json/cbor	Manufacturer identifier
dev/hwv	PropertyAffordance	:dpt.version	if.d	json/cbor	Hardware version
dev/fwv	PropertyAffordance	:dpa.0.25 :dpt.version	if.d	json/cbor	Firmware version
dev/hwt	PropertyAffordance	:dpt.varString8859_1	if.d	json/cbor	Hardware type
dev/model	PropertyAffordance	:dpa.0.15 :dpt.utf8	if.d	json/cbor	Device model or order number

Resource Name	Interaction	rt & Data Types	if	Content-Format	Description
dev/sna	PropertyAffordance	:dpa.0.57 :dpt.value1Ucount	if.p	json/cbor	KNX Subnet Address
dev/da	PropertyAffordance	:dpa.0.58 :dpt.value1Ucount	if.p	json/cbor	KNX Device Address
dev/hname	PropertyAffordance	:dpt.varString8859_1	if.p	json/cbor	Device hostname
dev/ipv6	PropertyAffordance	:dpt.ipv6	if.p	json/cbor	Device ipv6 addresses.
dev/fid	PropertyAffordance	:dpt.value8Ucount	if.p	json/cbor	KNX Fabric ID
dev/iid	PropertyAffordance	:dpt.value8Ucount	if.p	json/cbor	KNX Installation ID
dev/port	PropertyAffordance	:dpt.value2Ucount	if.p	json/cbor	Standard/default port number for unicast comm.
dev/mport	PropertyAffordance	:dpt.value2Ucount	if.p	json/cbor	Standard/default port number for multicast comm.
dev/pm	PropertyAffordance	:dpa.0.54 :dpt.binaryValue	if.p	json/cbor	Programming Mode
Application Program					
ap		:fb.3	if.ll	link-format	
ap/pv		:dpa.3.13 :dpt.programVersion	if.d	cbor	Application Program version
SW Update (PUSH)					
swu		:fb.swu	if.ll	link-format	
a/swu	ActionAffordance		if.swu	octet-stream	Package file (firmware update PUSH)
swu/pkgname	PropertyAffordance	:dpt.varString8859_1	if.swu if.d	cbor	Package file name
swu/pkgbytes	PropertyAffordance	:dpt.value4Ucount	if.swu	cbor	Transmitted and stored package bytes
swu/pkgv	PropertyAffordance	:dpt.version	if.swu if.d	cbor	Package version
swu/update	Action Affordance	:dpt.timePeriodSecZ	if.swu	cbor	Install package
swu/state	PropertyAffordance	:dpt.dldState	if.swu if.d	cbor	Update status
swu/result	PropertyAffordance	:dpt.updateResult	if.swu if.d	cbor	Update error
swu/lastupdate	PropertyAffordance	:dpt.varString8859_1	if.swu if.d	cbor	Last successful package update
swu/method	PropertyAffordance	:dpt.transferMethod	if.swu if.d	cbor	Download method
swu/protocol	PropertyAffordance	:dpt.protocols	if.swu if.d	cbor	Download protocol

Resource Name	Interaction	rt & Data Types	if	Content-Format	Description
Device Linking					
p	PropertyAffordance		if.b if.c	cbor/json	Parameter and diagnose properties
fp/g	PropertyAffordance		if.b if.c	cbor	Group object configuration
fp/r	PropertyAffordance		if.b if.c	cbor	Recipient configurations
fp/p	PropertyAffordance		if.b if.c	cbor	Publisher configuration
a/lsm	ActionAffordance		if.c	cbor	Device Load state Machine
S-Mode Messaging					
.knx	ActionAffordance		if.g.s	cbor	KNX S-Mode Group Notification Input
.knx	EventAffordance		if.g.s	cbor	KNX S-Mode Group Notification Output
Heat Valve Actuator					
hva		:fb.352	if.ll	link-format	
p/hva/actpossetpheatstagea	PropertyAffordance	:dpa.352.51 :dpt.scaling	if.i	json/cbor	Actuator Position Set Point Heat Stage A
p/hva/actposheatstagea	PropertyAffordance	:dpa.352.55 :dpt.scaling	if.o	json/cbor	Actuator Position Heat Stage A
User HVAC Display					
f/uhd		:fb.390	if.ll	application-link	
p/uhd/temproomsetpabseff	PropertyAffordance	:dpa.390.59 :dpt.valueTemp	if.p	json/cbor	Absolute Effective Room Temperature Set Point
Room Temperature Sensor					
f/rts		:fb.321	if.ll	link-format	
p/rts/temproom	PropertyAffordance	:dpa.321.51 :dpt.valueTemp	if.o	json/cbor	Measured Room Temperature
Manufacturer-specific					
f/devtype/	PropertyAffordance		if.ll	link-format	
p/devtype/batterylow	PropertyAffordance	:dpt.binaryValue	if.d	json/cbor	Battery Status
p/devtype/inlinkmargin	PropertyAffordance	:dpt.value1Ucount	if.d	json/cbor	In Link Margin
p/devtype/batteryuptime	PropertyAffordance	:dpt.timePeriodHrs	if.d	json/cbor	Uptime Battery
p/devtype/wakeupcycle	PropertyAffordance	:dpt.timePeriodSec	if.p	json/cbor	Wake Up Cycle Controller

6.2 DEVICE CONFIGURATION EXAMPLE

6.2.1 Full Download Example

In the first step, the MaC checks the device compatibility with the product database.

STEP 1: Check MaC DB entry compatibility with the KNX IoT device.

REQ:
POST coap://{ipv6-unicast}/dev/hwt
(OSCORE(code(GET), kid(<osc-id>)))

RES:
2.05 CONTENT (Content-Format: application/cbor (60))
Payload:
{
 1: "0123AB" //value: "01234AB"
}

Continue with the next steps only if the compatibility check is successful.

STEP 2: Set Device Load State Machine to "unload".

REQ:
POST coap://{ipv6-unicast}/a/lsm
(Content-Format: application/cbor (60), OSCORE(code(POST), kid(<osc-id>))
Payload:
{
 2: 4 //cmd: unload
}

If the KNX IoT device supports the intermediate state "unloading", then it may answer with the state "unloading". The MaC SHALL NOT continue before the KNX IoT has reached the state "unloaded".

STEP 3: Set Device Load State Machine to "loading".

REQ:
POST coap://{ipv6-unicast}/a/lsm
(Content-Format: application/cbor (60), OSCORE(code(POST), kid(<osc-id>)))
Payload:
{
 2: 1 //cmd: startloading
}

The MaC writes Function Point Table items to the KNX IoT device (see clause 2.6.7, "Creating, Updating, and Deleting Resources").

STEP 4: Write parameter configuration.

REQ:
POST coap://{ipv6-unicast}/p
(Content-Format: application/cbor (60), OSCORE(code(POST), kid(<osc-id>)))
Payload:

```
[
 {
  11: "p/rts/defaultsetpoint",
  1: 21,
 },
 {
  11: "p/rts/availablemodes",
  1: [1, 2, 3]
 },
 …
]
```

3237

STEP 5: Write Group Object Table.

REQ:
POST coap://{ipv6-unicast}/fp/g
(Content-Format: application/cbor (60), OSCORE(code(POST), kid(<osc-id>)))
Payload:

```
[
  {
    0: 13,                    //id
    11: "/ldsb1/soo",         //href
    7: [2305, 2401],          //ga
    8: 216                    //cflag: 0b11011000
  },
  {
    0: 14,
    11: "/LDSB1/RSC",
    7: [2306],
    8: 64                     //cflag: 0b01000000
  },
  {
    0: 15,
    11: "/ldsb2/soo",
    7: [2307, 2401],
    8: 216                    //cflag: 0b11011000
  },
  {
    0: 16,
    11: "/ldsb2/rsc",
    7: [2308],
    8: 64                     //cflag: 0b01000000
  }
]
```

3238

STEP 6: Write Function Point Publisher Table.

REQ:
POST coap://{ipv6-unicast}/fp/p
(Content-Format: application/cbor (60), OSCORE(code(POST), kid(<osc-id>)))
Payload:

```
[
  {
    0: 11,                              //id
    12: <publisher's IA>,               //ia
    7: [2305, 2306, 2307, 2308]         //ga
  },
  {
    0: 12,                                                              //id
    10: "coap://<IP multicast, unicast address or fqdn>:<port>/<path>", //url
    7: [2305, 2306, 2307, 2308]                                         //ga
  }
]
```

3239

STEP 7: Write Function Point Recipient Table.

REQ:
POST coap://{ipv6-unicast}/fp/r
(Content-Format: application/cbor (60), OSCORE(code(POST), kid(<osc-id>)))
Payload:

```
[
  {
    0: 9,                               //id
    12: <recipient's IA>,               //ia
    7: [2305, 2306, 2307, 2308]         //ga
  },
  {
    0: 10,                                                              //id
    10: "coap://<IP multicast, unicast address or fqdn>:<port>/<path>", //url
    7: [2305, 2306, 2307, 2308]                                         //ga
  }
]
```

3240

STEP 8: Write the Application Program version.

REQ:
POST coap://{ipv6-unicast}/ap/pv
(Content-Format: application/cbor (60), OSCORE(code(PUT), kid(<osc-id>)))
Payload:

```
{
 1: [250, 825, 16]
}
```

3241

STEP 9: Set Device Load State Machine to "loaded".

REQ:
POST coap://{ipv6-unicast}/a/lsm
(Content-Format: application/cbor (50), OSCORE(code(POST), kid(<osc-id>)))
Payload:
```
{
  2: 2     //cmd: loadcomplete
}
```

If the intermediate state "LoadCompleting" is supported by the KNX IoT device, then it may answer with the state "LoadCompleting". The MaC SHALL NOT continue before the KNX IoT device has reached the state "loadcomplete".

STEP 10: Read the configuration fingerprint of the written resources.

REQ:
POST coap://{ipv6-unicast}/.well-known/knx/f
(OSCORE(code(GET), kid(<osc-id>)))

RES:
2.05 CONTENT (Content-Format: application/cbor (60))
Payload:
```
{
  1: 132768923564196
}
```

The retrieved fingerprint MAY be stored by the MaC for a future decision about a differential download.

Finally, the MaC MAY restart the device to cause a reset of the communication system (e.g., device group table) in the before-configured device (see clause 2.5.5.3.3, "(Basic) Restart Command").

STEP 11: Restart device (optional).

REQ:
POST coap://{ipv6-unicast}/.well-known/knx
(Content-Format: application/cbor (60), OSCORE(code(POST), kid(<osc-id>)))
Payload:
```
{
  2: "restart"
}
```

6.2.2 Partial Download Example

In the first step, the MaC checks the device compatibility with the product database.

STEP 1: Check the MaC DB entry compatibility of the KNX IoT device.

REQ:
POST coap://{ipv6-unicast}/dev/hwt
(OSCORE(code(GET), kid(<osc-id>)))

RES:
2.05 CONTENT (Content-Format: application/cbor (60))
Payload:
{
 1: "0123AB" //value: "0123AB"
}

Continue with the next steps only if the compatibility check is successful.

STEP 2: Check if the KNX IoT device is in a known state.

REQ:
POST coap://{ipv6-unicast}/.well-known/knx/f
(OSCORE(code(GET), kid(<osc-id>)))

RES:
2.05 CONTENT (Content-Format: application/cbor (60))
Payload:
{
 1: 132768923564196
}

If the MaC knows the device configuration fingerprint value, then the MaC MAY continue with a differential download. Otherwise, the MaC SHALL start with the full download and continue with STEP 2.

STEP 3: Set Device Load State Machine to "loading".

REQ:
POST coap://{ipv6-unicast}/a/lsm
(Content-Format: application/cbor (60), OSCORE(code(POST), kid(<osc-id>)))
Payload:
{
 2: 1   //cmd: startloading
}

STEP 4: Write parameter configuration (conditional: only if changes there).

REQ:
POST coap://{ipv6-unicast}/p
(Content-Format: application/cbor (60), OSCORE(code(POST), kid(<osc-id>)))
Payload:

```
[
 {
  11": "p/rts/defaultsetpoint",  //href
  1: 21,                         //value
 },
 {
  11: "p/rts/availablemodes",    //href
  1: [1, 2, 3"]                  //value
 },
 …
]
```

The MaC MAY write only changed parameters (identified by href).

STEP 5: Write Group Object Table (conditional: only if changes there).

REQ:
POST coap://{ipv6-unicast}/fp/g
(Content-Format: application/cbor (60), OSCORE(code(POST), kid(<osc-id>)))
Payload:

```
[
  {
    0: 13,                       //id
    11: "/ldsb1/soo",            //href
    7: [2305, 2401],             //ga
    8: 216                       //cflag: 0b11011000
  },
  {
    0: 14,
    11: "/ldsb1/rsc",
    7: [2306],
    8: 64                        //cflag: 0b01000000
  },
  {
    0: 15,
    11: "/ldsb2/soo",
    7: [2307, 2401],
    8: 216                       //cflag: 0b11011000
  },
  {
    0: 16,
    11: "/ldsb2/rsc",
    7: [2308],
    8: 64                        //cflag: 0b01000000
  }
]
```

The MaC MAY write only changed group objects (identified by "id").

STEP 6: Write Function Point Publisher Table (conditional: only if changes there).

REQ:
POST coap://{ipv6-unicast}/fp/p
(Content-Format: application/cbor (60), OSCORE(code(POST), kid(<osc-id>)))
Payload:

```
[
  {
    0: 11,                                  //id
    12: "<publisher's IA>",                 //ia
    7: [2305, 2306, 2307, 2308]             //ga
  },
  {
    0: 12,                                  //id
    10: "coap://<IP multicast, unicast address or fqdn>:<port>/<path>",//url
    7: [2305, 2306, 2307, 2308]             //ga
  },
  {
    0: 13                                   //id
  }
]
```

The MaC MAY write only changed table entries (identified by "id"). Entries with a new "id" will create a new entry.

STEP 7: Write Function Point Recipient Table (conditional: only if changes there).

REQ:
POST coap://{ipv6-unicast}/fp/r
(Content-Format: application/cbor (60), OSCORE(code(POST), kid(<osc-id>)))
Payload:

```
[
  {
    0: 9,                                   //id
    12: "<recipient's IA>",                 //ia
    7: [2305, 2306, 2307, 2308]             //ga
  },
  {
    0: 10,                                  //id
    10: "coap://<IP multicast, unicast address or fqdn>:<port>/<path>",//url
    7: [2305, 2306, 2307, 2308]             //ga
  }
]
```

The MaC MAY write only changed table entries (identified by "id"). Entries with a new "id" will create a new entry.

STEP 8: Write the Application Program version.

REQ:
POST coap://{ipv6-unicast}/ap/pv
(Content-Format: application/cbor (60), OSCORE(code(PUT), kid(<osc-id>)))
Payload:
```
{
  1: [250, 825, 16]
}
```

3271

STEP 9: Set Device Load State Machine to "loaded".

REQ:
POST coap://{ip-unicast}/a/lsm
(Content-Format: application/cbor (60), OSCORE(code(POST), kid(<osc-id>)))
Payload:
```
{
  2: 2     //loadcomplete
}
```

3272
3273 If the intermediate state "loadcompleting" is supported by the KNX IoT device, it MAY answer with the
3274 state "loadcompleting". The MaC SHALL NOT continue before the KNX IoT device has reached the
3275 state "loadcomplete".

STEP 10: Read the configuration fingerprint of the written resources.

REQ:
POST coap://{ip-unicast}/.well-known/knx/f
(OSCORE(code(GET), kid(<osc-id>)))

RES:
2.05 CONTENT (Content-Format: application/cbor (60))
Payload:
```
{
  1: 132768923564196
}
```

3276
3277 The MaC MAY store the retrieved checksum for a future decision about a differential download.

3278 Finally, the MaC MAY restart the device to cause a reset of the communication system (e.g., device
3279 group table) in the before-configured device.

STEP 11: Restart device (optional).

REQ:
POST coap://{ipv6-unicast}/.well-known/knx
(Content-Format: application/cbor (60), OSCORE(code(POST), kid(<osc-id>)))
Payload:
{
2: "restart"
}

6.3 DATA ENCRYPTION/DECRYPTION EXAMPLE

6.3.1 OSCORE Unicast

The following clauses describe the OSCORE message content from clause 3.6.5, "Message Processing".

6.3.1.1 PASE Example

The following figure shows how a PASE access token for the responder key is created and from where the message content is taken for decrypting an incoming message:

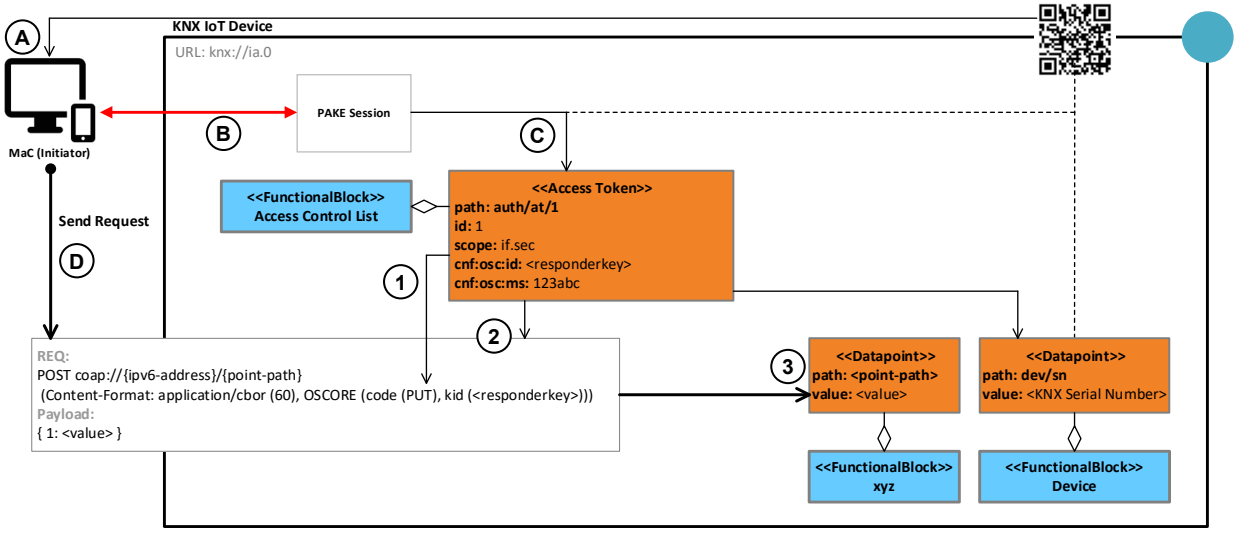


Figure 31 – PAKE-based OSCORE Credential Configuration Example

- A: The MaC gets the *KNX Serial Number*, for example, from a QR-Code.
- B: The MaC and the KNX IoT device derive a new shared secret for the PASE session (see 3.6.3).
- C: The KNX IoT device creates a new access token valid for this PASE session. The derived shared secret for the PASE session is temporarily stored in an access token. (cnf:osc:ms)
- D: The MaC uses the secret for encoding the OSCORE message (unicast).
- 1: The KNX IoT device searches the access token with the same OSCORE input material identifier (osc-id = cnf:osc:id) as the OSCORE "kid" of the message (<rkey>).
- 2: The OSCORE cryptographic parameters from the corresponding access token are used for message decryption.
- 3: If the client has permissions (see scope), the KNX IoT device writes the value to the corresponding Point.

6.3.1.2 Unicast Read/Write Point Example

The next figure depicts a request/response message flow between a KNX IoT client and server device:

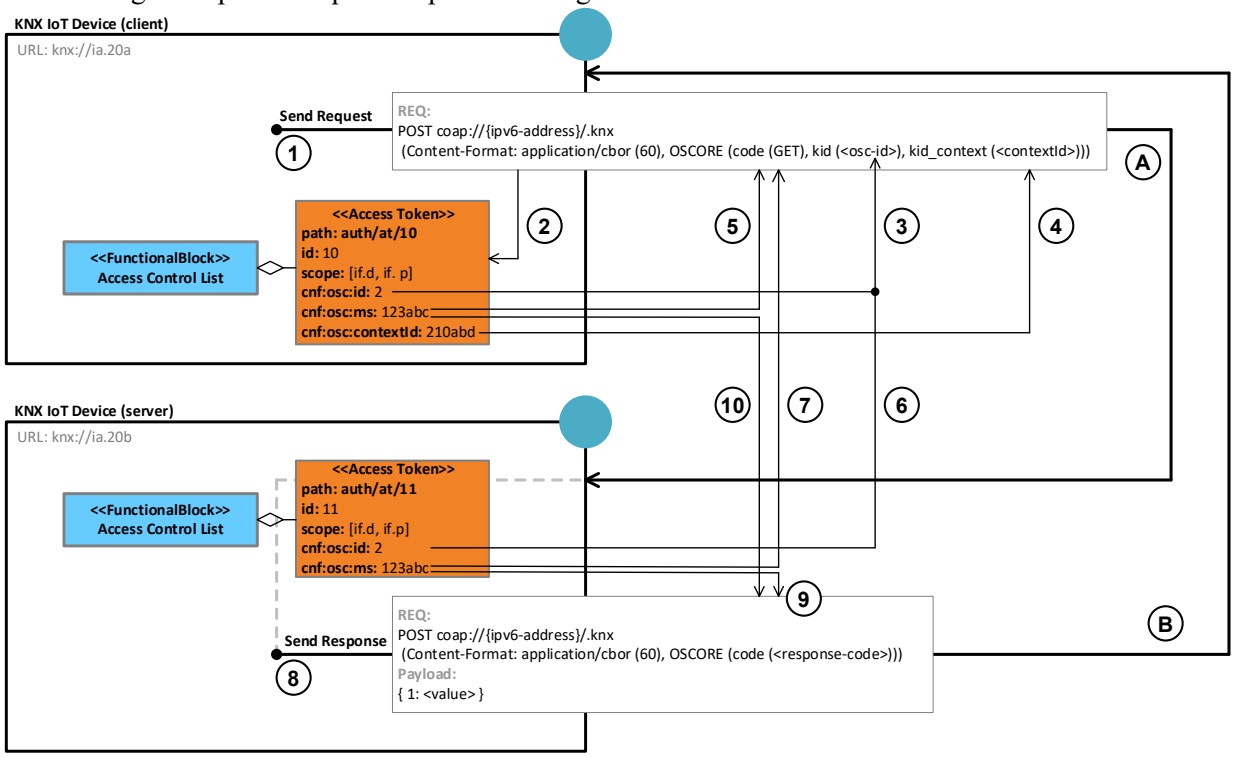


Figure 32 –Unicast OSCORE Request/Response Example

- 1: The KNX IoT device (client) wants to send a request to a KNX IoT server.
- 2: The KNX IoT device queries the Function Point Table containing the corresponding access token.
- 3: The KNX IoT device prepares the request message and uses the OSCORE input material identifier (osc-id = cnf:osc:id) of the access token as the message OSCORE "kid".
- 4: The KNX IoT device adds the OSCORE "contextId" from the access token to the message OSCORE "kid_context" to avoid message duplications.
- 5: The OSCORE cryptographic parameters are used for message encryption.
- A: The KNX IoT device sends the encrypted message to the server.
- 6: The KNX IoT device (sever) queries the access control list and searches the corresponding access token item containing the same OSCORE input material identifier (osc-id = cnf:osc:id) as in the OSCORE "kid" in the message.
- 7: The OSCORE cryptographic parameters are used for message decryption.
- 8: The server checks client permissions and replies accordingly.
- 9: The OSCORE cryptographic parameters are used for message encryption. The KNX IoT device (server) uses an empty byte string for the OSCORE sender ID (kid = h").
- B: The KNX IoT device sends the encrypted response message to the client.
- 10: The KNX IoT device (client) queries pending request sessions and the corresponding credentials for decrypting the unicast response.

6.3.2 OSCORE Multicast

6.3.2.1 Multicast S-Mode Example

The following figure explains from which Points message content is taken before encrypting and sending an OSCORE message:

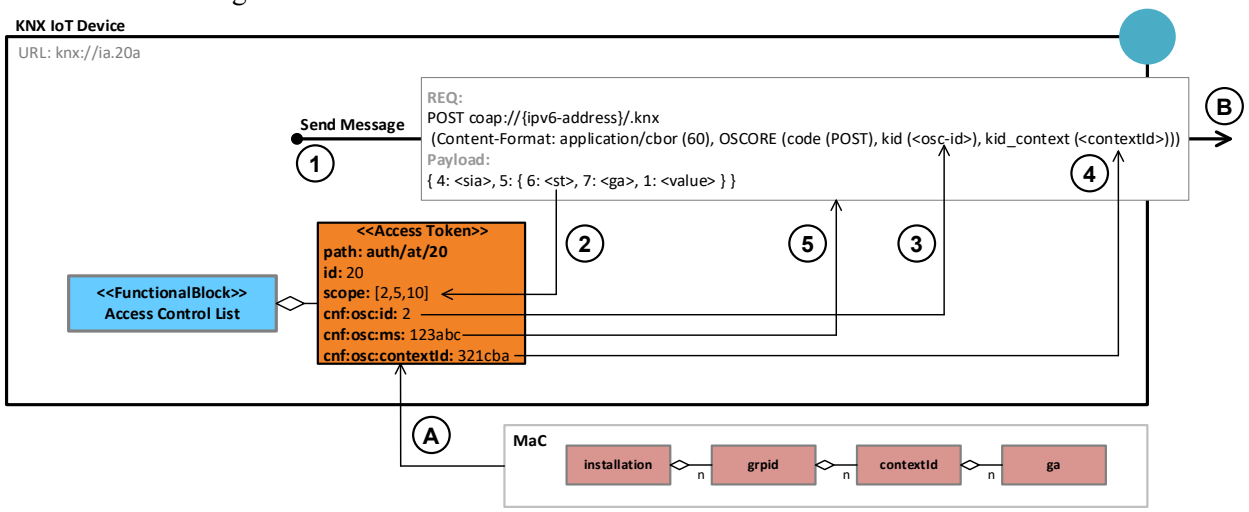


Figure 33 –Multicast OSCORE S-Mode Example

- A: The MaC configures access tokens on a KNX IoT device. The access token contains the OSCORE credentials with the necessary information for message encryption and decryption. The same access token can be used for multiple *Group Addresses* ("ga") belonging to the same OSCORE input material identifier (osc-id = cnf:osc:id).
- 1: The KNX IoT device wants to send a message because a value has changed on a Point configured in the Group Object Table. After checking the Function Point Table, the message gets the group address ("ga").
- 2: The Function Point Publisher or Recipient table item with the corresponding *Group Addresse* ("ga") contains a reference to the corresponding access token in the access control list.
- 3: The OSCORE input material identifier (osc-id = cnf:osc:id) of the access token is used for the OSCORE message "kid".
- 4: The KNX IoT device adds the "kid_context" in the multicast message OSCORE "kid_context" to avoid message duplications.
- 5: The OSCORE cryptographic parameters are used for message encryption.
- B: The message sequence number SHALL be stored on the receiving device for each security context identified by the "kid" and the "kid_context". The Recipient handles the incoming message as any other request (see also 6.3.1.2) but usually does not reply.

This page is reserved to list all endnotes.

[i] JSON is used to denote contents in this paper. In practice, CBOR encoding is used.